

图灵原创



Unity 3D 游戏开发

宣雨松○编著



人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明



宣雨松，网名雨松MOMO。首款入驻美国苹果店数字体育产品GolfSense Unity 3D主程，CSDN博客专家，51CTO博客之星，51CTO移动开发论坛版主，Unity圣典开发论坛版主，5年以上移动游戏与软件开发经验。曾经领导与参与的游戏项目包括：GolfSense、新少林寺、天降少女、游龙戏凤西门庆等。个人独立博客地址：http://www.xuanyusong.com/。

图书在版编目（CIP）数据

Unity 3D游戏开发 / 宣雨松编著. -- 北京 : 人民邮电出版社, 2012.6
（图灵原创）
ISBN 978-7-115-28381-8

Ⅰ. ①U… Ⅱ. ①宣… Ⅲ. ①游戏程序—程序设计
Ⅳ. ①TP311.5

中国版本图书馆CIP数据核字(2012)第110826号

内 容 提 要

本书通过实例详细介绍了如何使用 Unity 进行游戏开发，书中先简要介绍了 Unity 环境搭建、编辑器和 GUI 游戏界面相关的知识，接着介绍了如何使用 C# 和 JavaScript 构建游戏脚本，添加树、草、石头等模型以及键盘事件、鼠标事件和 3D 模型动画相关的内容，然后介绍了持久化数据、音频与视频播放等内容，最后以一款第一人称射击类游戏为原型，向读者详细介绍游戏制作的整个过程。

本书适合具备一些 JavaScript 与 C# 语言基础，并且想快速入门 Unity 3D 游戏开发的人员阅读。

图灵原创
Unity 3D游戏开发

◆ 编著　宣雨松
责任编辑　王军花
◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街14号
邮编　100061　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷
◆ 开本：800×1000　1/16
印张：21.75
字数：508千字　2012年 6 月第 1 版
印数：1 – 4 000册　2012年 6 月北京第 1 次印刷

ISBN 978-7-115-28381-8

定价：59.00元

读者服务热线：(010)51095186转604　印装质量热线：(010)67129223
反盗版热线：(010)67171154

序

互联网和移动设备的火速发展，改变着整个世界的步伐，游戏行业也紧随着快速发展。对于各种各样的设备终端，传统的纯代码开发已经显得非常吃力，那么我们如何应对这个局面呢?

一款产自丹麦、由Unity Technologies开发的游戏引擎Unity，带着强大的跨平台开发等特性来到了我们面前，组件式的开发让你告别枯燥乏味的纯代码式开发。与其他游戏引擎最显而易见的特点就是，一次开发即可轻松部署到Windows、Mac、iOS、Android、Wii、PS3等平台，告别以往高难度的、耗时的跨平台游戏开发，使快速的、高质量的游戏开发成为可能。

Unity具有高度优化的图形渲染管道，无论2D游戏还是3D游戏，均可达到美轮美奂的画面效果。它支持所有主要的文件格式引入，使得美术工作者在自己熟悉的创作工具中尽情发挥，而不必因为文件格式兼容问题影响效率和效果。它内建NVIDIA PhysX物理引擎，让你轻松实现现实中生动的互动效果。此外，它支持JavaScript、C#和Boo三种脚本语言，新增的强大的寻路系统、焕然一新的Shuriken粒子系统、“镜之边缘”所使用的光影烘焙系统、改进的遮挡拆切和LOD系统等都是提高你游戏质量、节省时间的得力工具。

那么我们该如何学习并使用Unity开发游戏呢？目前，Unity虽然有大量的英文资料，但中文资料还非常少，令一些初学者怯步。不过本书的面世解决了这一问题，无论初学者还是已经接触过Unity的使用者，都将通过本书学到相关知识，提高使用Unity开发游戏的能力。本书详尽介绍了Unity的安装、使用及深入开发等，并包含相应的实例来巩固知识点，是快速入门及提高Unity技术的必备书。愿本书能给我们大家带来越来越多由Unity开发的优秀游戏！

Unity资深用户四角钱

前　言

近年来，游戏行业出现了前所未有的震荡期，各种平台的涌现使得行业内部的竞争愈演愈烈。前几年，要想制作好的游戏，肯定就需要强大硬件的支持，所以大部分3D游戏都出现在PC或PS3、Xbox等专业游戏主机上，其他平台则由于受硬件条件的限制而无法制作出较好的游戏。

目前基于iOS、Android、Windows Phone 7等移动平台的智能手机迅速崛起，它们的硬件配置已经得到大幅提升，目前的硬件条件已经达到了几年前的水平，在移动平台上制作3D游戏已不再是梦想。此外，Flash与HTML5也开始对网页中的图形加速渲染，这也使得网页游戏得到了迅速发展。在这场浩浩荡荡的战争中，鹿死谁手迄今无人可知，套用游戏玩家的一句老话“没有最强的职业，只有最强的玩家”，一切我们只好拭目以待。

由于平台之间激烈的竞争，游戏开发商在制作游戏时非常头大，因为不知道选择从哪种平台入手。平台的不同就意味着开发方式也截然不同，所以“跨平台游戏开发”这样的字眼近年来也慢慢出现在我们的视野当中。跨平台开发的好处是一次编码多平台适用，只需要花人力、物力和财力制作一遍，就可适用全部游戏平台，这样将大大节约开发成本。

目前市面上的跨平台游戏引擎已经有好几款，其中最专业、最稳定、效率最高并且支持游戏平台最多的就是Unity。目前它的最新版本为Unity 3.5，可横跨9种主要游戏平台，包括Web平台、PC平台、Mac平台、iOS平台、Flash平台、Android平台、Xbox 360平台、PS3平台和Wii平台。无论是电脑游戏、网游游戏、手机游戏、主机游戏、单机游戏还是网络游戏，都可以使用Unity轻松实现跨平台游戏开发。

现在网络游戏已不局限于电脑终端，手机与网页终端的网络游戏也迅速走进玩家的世界。分析师预测，未来的网络游戏肯定是跨平台网游，玩家不仅可以在电脑上，还可以在手机或者网页甚至在掌机中去玩游戏，这是多么酷的一件事啊，所以Unity将肩负重任。

本书主要从初学者入门的角度去讲解Unity游戏开发，无论读者是转行学习也好，入门学习也罢，甚至是毫无任何编程经验的人员，都可以阅读本书，并且可以让读者快速学会如何使用Unity制作3D游戏。本书将以两种语言去讲解游戏脚本的编写，由于JavaScript语言更适合初学者，所以入门阶段我们将使用JavaScript语言介绍，而在进阶阶段将使用C#语言。为方便读者学习，书中每一章都含有丰富的游戏实例与源代码。最后一章以一个第一人称射击类3D游戏为实例充分向读者介绍游戏实战开发的过程。

阅读本书

书中所有例子的源代码都可以在图灵网站（www.ituring.com.cn）本书主页免费注册下载，并且下载的源码按章编号，如图0-1所示。查看源码前，请确保Unity已经安装在本机当中，安装无误后按照图中所示找到章节源码对应的游戏场景文件，双击该场景文件即可打开游戏工程，继而查看阅读。

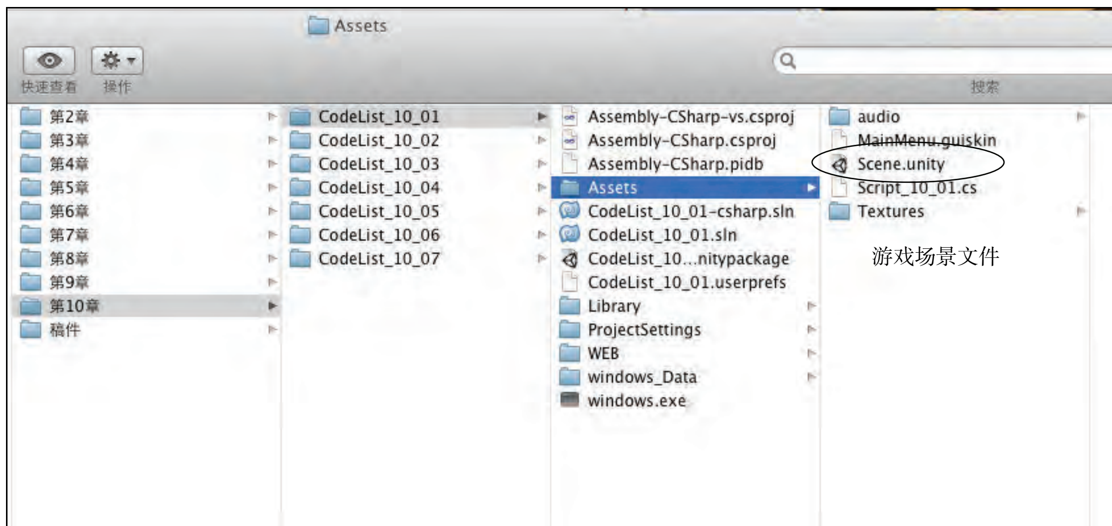


图0-1 查看源码

由于Unity自带的脚本编辑器MonoDevelop无法正常显示中文，编辑器Unitron虽然可以显示中文，但是需要修改编码格式，而它无法修改默认打开时采用的编码格式，所以使用脚本编辑器直接打开脚本文件，都会出现中文乱码的情况。为了解决这个问题，读者可以先以记事本格式打开示例游戏中的脚本，然后复制至Unity自带的另一款编辑器Unitron当中查看，接着在Unitron中修改编码格式。在用JavaScript语言编写的脚本中，请使用UTF-8编码格式，而在C#语言编写的脚本中，请使用UTF-16编码格式。

由于本书是在Mac OS下讲解Unity开发，如果读者采用Windows操作系统的话，查看源码的方式会与Mac OS有点区别，并且按照上述的方法是无法查看源码的。为了方便读者在Windows操作系统下查看游戏源码，笔者已将游戏工程封装成自定义游戏包放置在工程根目录下，读者只需导入工程对应的游戏包即可查看源码。首先打开Windows下对应的Unity，在导航菜单栏中选择“Assets”→“Import Package”→“Custom Package”菜单项，如图0-2所示。

在源码的根目录中寻找需要导入的自定义游戏包，如图0-3所示，该游戏包的文件名后缀为unitypackage，选择后打开即可。此外，读者也可将游戏包拖曳到其他目录（比如桌面）当中，然后选择该游戏包，双击它也可导入工程。

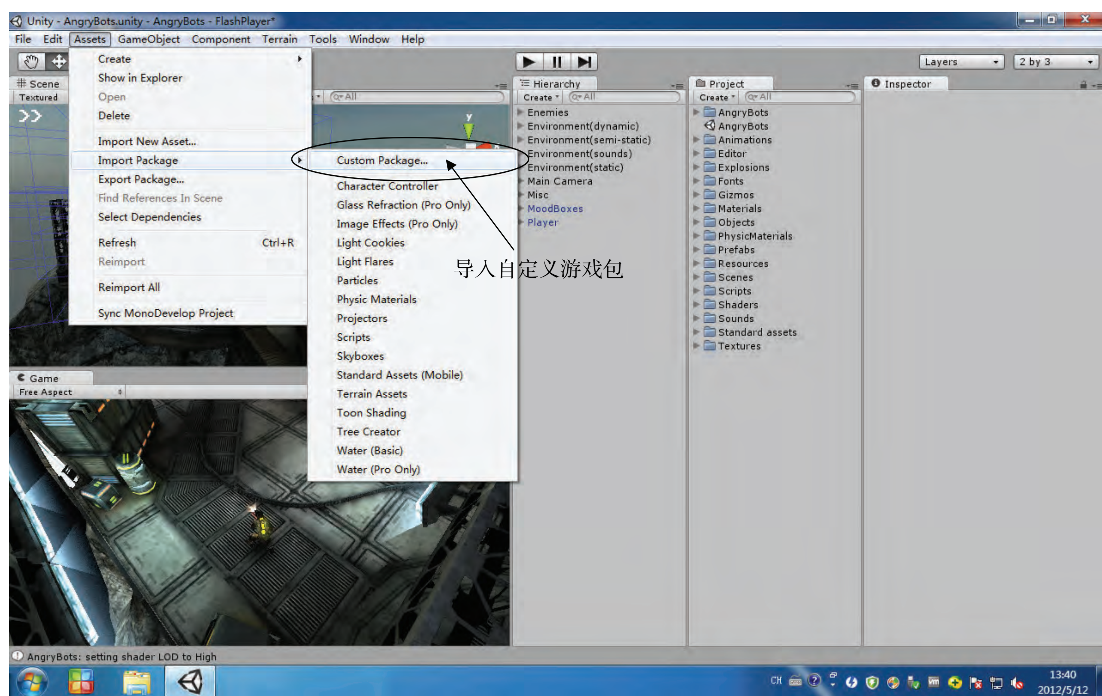


图0-2　导入程序包

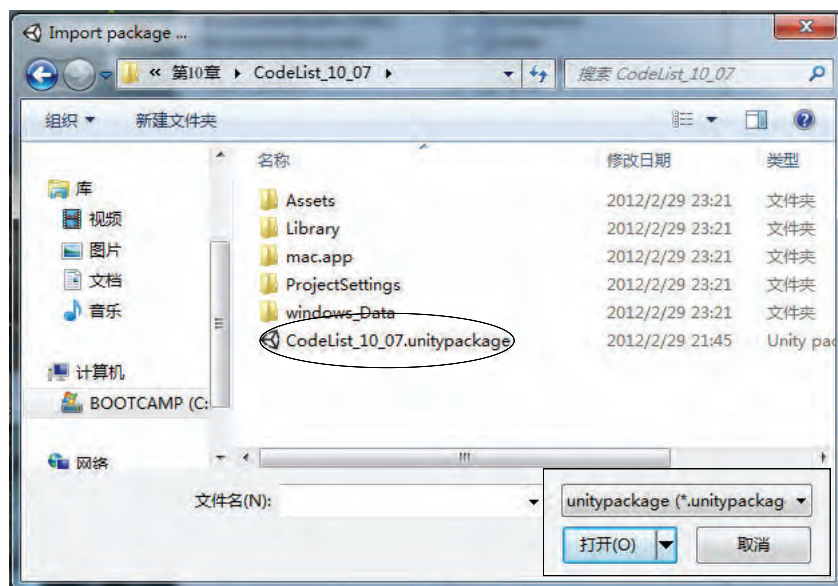


图0-3　寻找需要导入的自定义程序包

如图0-4所示，自定义游戏包中的资源已经出现在列表当中，点击右下角的“Import”按钮即可完成导入，然后运行游戏即可看到效果。

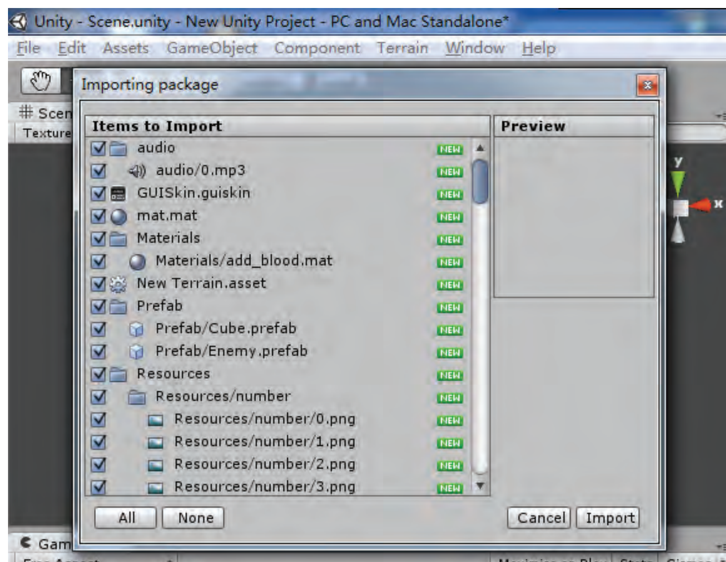


图0-4　资源出现在列表当中

致谢

我从事移动终端开发已有5个年头，从早期的第一代诺基亚智能手机到Android和iOS，从软件开发到游戏开发，从2D游戏到3D游戏，这对一个程序员来说，说短不短说长也不长。在这期间，我首先需要感谢一个人，他就是我的师父王嘉鑫，这些年来他让我从一名刚毕业的学生磨练成为一名真正的工程师，滴水之恩我将涌泉相报；第二个需要感谢的就是一名大姐姐王璐，她在我迷茫的时候为我指了一条路，正是因为她我才有机会走进了游戏行业；第三个需要感谢的就是我的女友和家人，正是他们的理解与支持，我才取得今天的成绩；还要感谢公司里一起经常通宵加班的好同事、好兄弟，希望我们大家今后能继续一如既往地好好工作，为了人生的理想一起奋斗！

此外，我必须感谢图灵公司的杨海玲与小花，正是因为她们的鼓励，我才有动力去写这本书，感谢她们教会了我很多写作技巧与一些重要的写作建议，感谢她们仔细审阅了我的书稿，确保这本书的整体质量；还要感谢我的同事孙磊和张尧为本书制作了很多好看的美术资源。最后，祝图灵公司越做越好，为祖国的IT人才培养贡献出伟大的力量。

2012年2月29日
宣雨松

目　　录

第 1 章 基础知识

Unity是一款3D跨平台次世代游戏引擎，“Unity”一词的中文解释为“团结”，好比集合所有人的力量一起来完成一件伟大的巨作一般，这款游戏引擎以其强大的跨平台特性与绚丽的3D渲染效果而闻名出众。该款游戏引擎的开发商是大名鼎鼎的Unity Technologies，近年来该公司处于飞速发展当中。起初Unity的版本为1.0.0，它只可部署在Mac OS下并且只能制作iPhone中的游戏。随着它不断发展与壮大，目前Unity的版本已经升级至3.5，可同时部署在Mac OS与Windows两种操作系统之上，横跨的主流游戏平台高达9种。而且其3D渲染效果也得到了大幅度提升。

目前全球的Unity注册用户已经超过6000万，国内首款Unity 3D PC网游《将魂》已震撼面市，开启了网络游戏的新纪元。此外，大量的Unity 3D网页游戏也涌现在我们的视线当中。在移动方面，苹果的App Store中有1500多款游戏是使用Unity进行开发的，Android平台中也有不少Unity制作的优秀游戏。总之，Unity近几年的蓬勃发展已经让它在游戏业内站稳脚跟，让越来越多的好游戏脱颖而出。未来，Unity还会融合更多炫酷的功能并且横跨更多的游戏平台，请大家拭目以待。

1.1 Unity 简介

Unity是一款标准的商业游戏引擎，而商业引擎的主要特点有收费、封闭源码和功能强大。

关于收费情况，Unity的使用费用非常昂贵，最便宜的普通版许可证也需要400美元，加强版本为1500美元。当然许可证的版本不同，引擎支持的功能也就截然不同。具有加强版许可证的引擎的很多强大功能是有普通版许可证的引擎所不具备的。但是从学习的角度来说，无须购买许可证同样可以进行，因为许可证更大的用处在于游戏制作完成后的打包与发布，不购买许可证制作的游戏是无法发售的，不过，单就学习而言，我们完全无须购买许可证，选择免费的Unity即可。

游戏引擎的开源与闭源是两种主要的趋势，它们之间各有利弊。如果采取开源形式，那么为了学习与钻研引擎，使用人数肯定会大幅度提高，缺点是因为已经将源代码赤裸裸地发放，所以第三方对源码加以修改，容易造成不劳而获的情况。而封闭源码则可以让引擎更加安全，并且有效地保护引擎的知识产权。Unity就是采取完全封闭源码的形式。

Unity引擎的功能非常强大，其中一个显著特点就是跨平台游戏开发。跨平台开发无疑为开发者节省了大量时间。平台之间的差异会直接影响到开发进度，比如屏幕尺寸、操作方式、硬件条件等的不同会给开发者造成巨大的麻烦，因为在不同的平台中开发者需要花更多时间去做平台之间的移植开发，而将大量时间浪费在这上面并不值得。Unity几乎为开发者完美地解决了这一

难题，将大幅度减少移植过程中一些不必要的麻烦，但使用它后也并非一点麻烦都不会产生。因为各平台的硬件条件是不同的，比如PC的硬件条件肯定会强于移动平台，所以开发者还需要针对不同的平台做一番取舍。

介绍完Unity的主要特点后，下面简要介绍一下开发环境。Unity可部署在Mac OS或Windows操作系统中，在这两种操作系统中，除了客户端操作习惯与界面有些差异以外，引擎自身的功能没有任何区别。学习Unity之前，请读者选择适合自己的操作系统。

1.2　下载与安装

Unity引擎官方的下载地址为http://unity3d.com/unity/download/。在撰写本书之时，Unity官方的最新版本为Unity 3.5，所以书中将主要以Unity3.5进行讲解。后续如果Unity版本进行了升级，读者亦可在Unity官网下载最新版本并结合本书进行学习。因为Unity支持向下兼容，所以在新版本中同样可以运行书中的所有游戏例子，这点请读者放心。

首先我们登录Unity官网的下载地址开始下载Unity 3.5的安装包。打开Unity官方下载网页（如图1-1所示）后，点击右侧的Download Unity 3.5按钮，开始下载Unity的程序安装包。下载时，官网会检测下载此安装包所使用的操作系统从而进入对应的下载页面，比如读者使用Mac OS操作系统，下载Unity时下载页面就是Mac OS版本，使用Windows操作系统时，下载Unity时下载页面就是Windows版本。当然，也可自行选择下载的Unity程序包，在图中右下方点击“Developing on Windows”链接可切换到Windows版本的下载页面，点击“Developing on Mac OS X”链接可切换到Mac OS版本的下载页面。

如图1-1所示，在Download Unity 3.5按钮下方，还有3个比较重要的链接，其含义如下所示。

- System Requirements：系统需求，开发环境的硬件需求。
- License Comparison：许可证对比，许可证版本之间的区别。
- Release Notes：版本发布说明，这里包含Unity所有历史版本的发布说明以及它们的下载地址。



图1-1　Unity下载页面

目前Unity支持在Mac OS与Windows两种操作系统下编写游戏，本节我们将向读者详细介绍如何在这两种操作系统下搭建Unity开发环境。下面我们先介绍如何在Mac OS中搭建Unity开发环境。

1. 在Mac OS下安装Unity

下载完毕后，可以看到Unity 3.5的安装包，双击它后即可进行Unity的安装。这里需要说明的是，在首次安装Unity时需要联网注册，只有注册成功后才可以使用Unity。启动Unity 3.5安装包后，程序将弹出Unity的注册向导界面，如图1-2所示。



图1-2　注册向导界面

在页面下方点击“Register”按钮后，此时将进入Unity激活界面，如图1-3所示。在激活界面中，首先需要选择激活方式。激活方式有联网激活（Internet activation）与手动激活（Manual activation）两种。联网激活适用于未购买Unity许可证的用户，而手动激活适用于已购买许可证的用户，这里我们选择“Internet activation”进行联网激活。

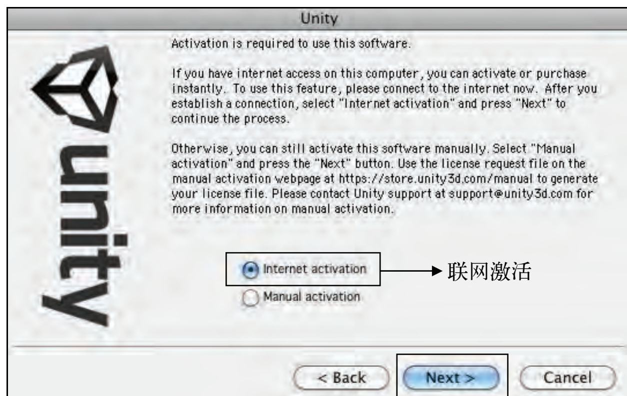


图1-3　Unity激活页面

点击“Next”按钮，程序会自动帮我们打开Unity联网激活的页面，如图1-4所示。为了完成激活，读者需要按照提示正确填写注册的相关信息，它们包括开发者邮箱与公司地址。内容填写完毕后，直接点击“Free”按钮免费激活Unity。

这里需要说明一下，Unity的试用期只有30天。我们知道Unity的许可证可分为普通版与加强版两种，普通版本只具备Unity的基本功能，而加强版本更为强大，比如增强了3D特效、特殊的光影效果、3D渲染特效等。然而30天的使用期限是针对于加强版本的，30天后将无法继续免费使用加强版本中的功能，需要缴费购买许可证，但是普通版本的功能仍然可以继续使用。关于普通版本与加强版本两者的详细区别，读者可点击“License Comparison”按钮进行查看。

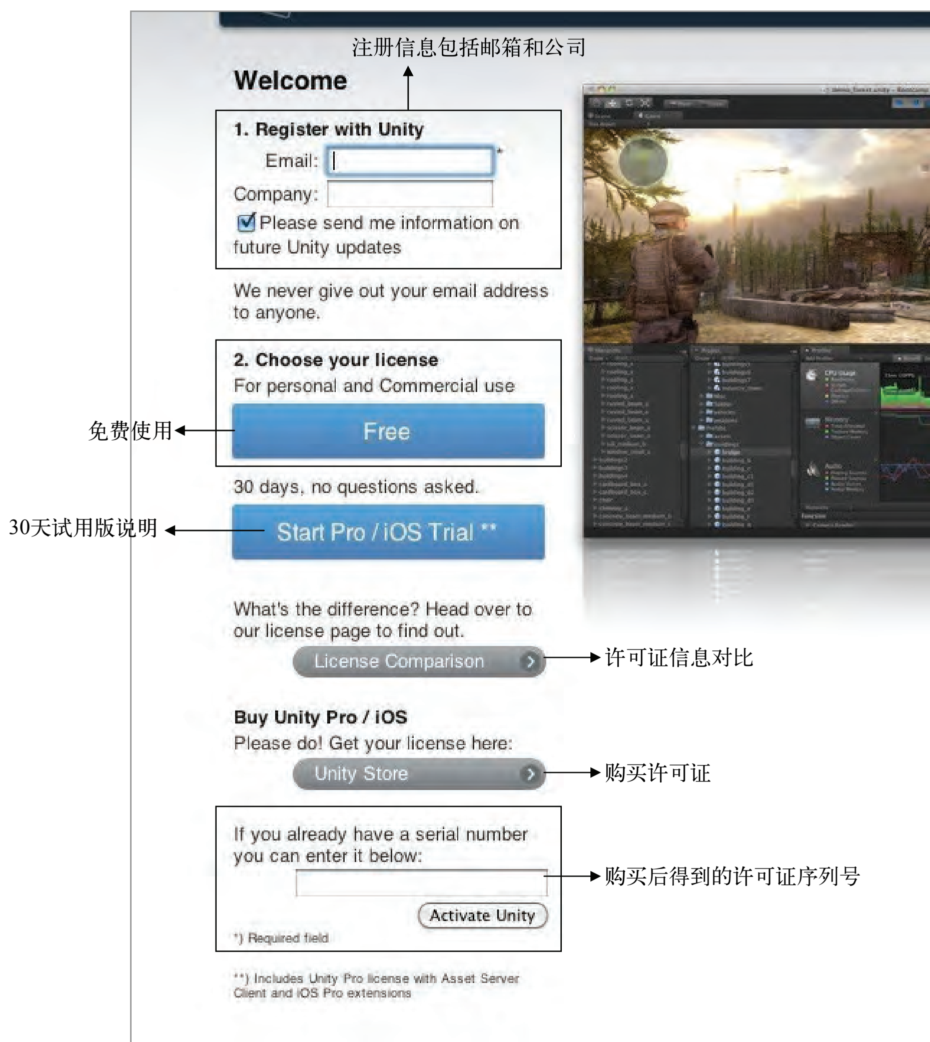


图1-4　联网激活页面

点击“Free”按钮后，程序将打开注册完成界面，如图1-5所示，这表示Unity的 Mac OS版本已经彻底在本机中注册完毕，然后在界面中点击下方的“Finish”按钮，开始我们Unity学习之旅。

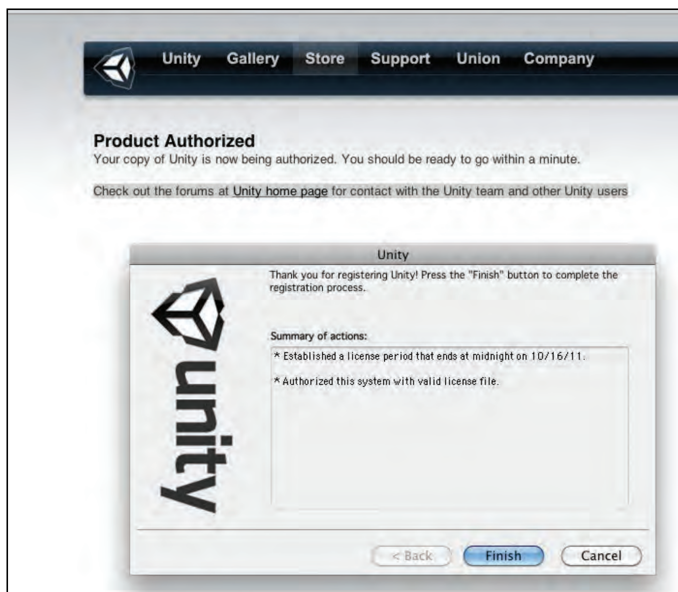


图1-5　注册完成

需要说明的是，Unity注册向导只会在本机第一次安装Unity时出现，注册完毕即表示安装成功。后续如果读者需要覆盖安装或者升级，只需打开Unity程序安装包，根据提示点击下一步即可。

2. 在Windows下安装Unity

在Windows下安装Unity与在Mac OS下安装有细微的差别。首先下载Unity 3.5 Windows版本，然后打开它开始安装。

在Windows中首次安装Unity同样需要注册，由于注册方法与Mac OS完全一样，这里就不再赘述。但是在Windows下注册完毕后，需要进行安装，具体操作如下：首先按照Mac OS中的注册方法在本机完成注册，然后程序将弹出安装Unity界面，如图1-6所示。

点击“Next”按钮，将进入安装说明界面，如图1-7所示。

安装前，请仔细阅读安装说明，确保无误后点击“I Agree”按钮继续安装，此时将进入Unity选择安装界面，如图1-8所示。除了安装Unity主程序外，还可选择性安装一些插件或工具。下面简单介绍一下这些安装组件。

- Unity：主程序，必须安装。
- Example Project：示例程序，可供用户参考。

- Unity Development Web Player：Web开发者安装包。
- MonoDevelop：脚本编辑器，强烈建议安装。

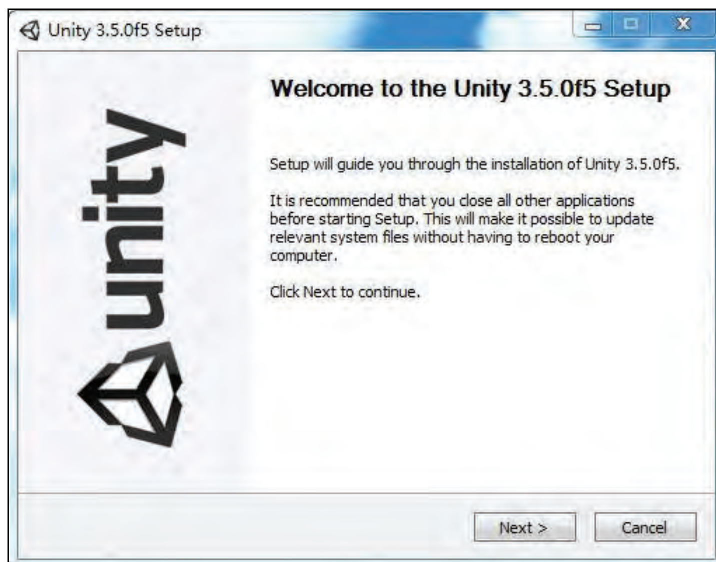


图1-6　开始安装

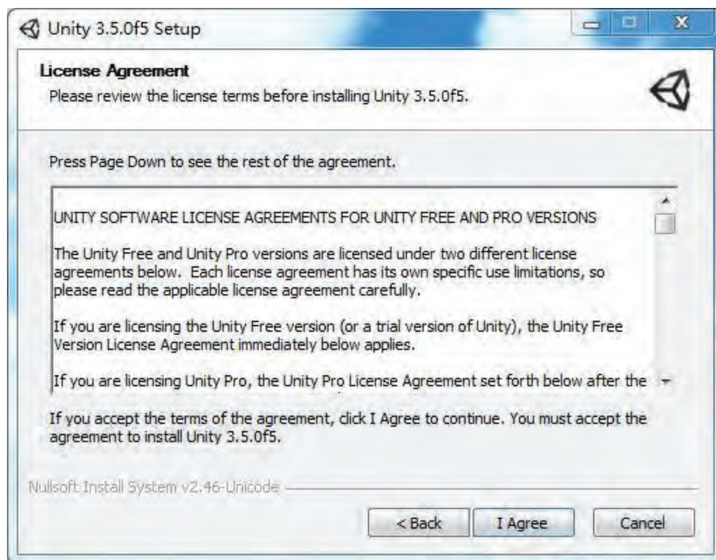


图1-7　安装说明界面

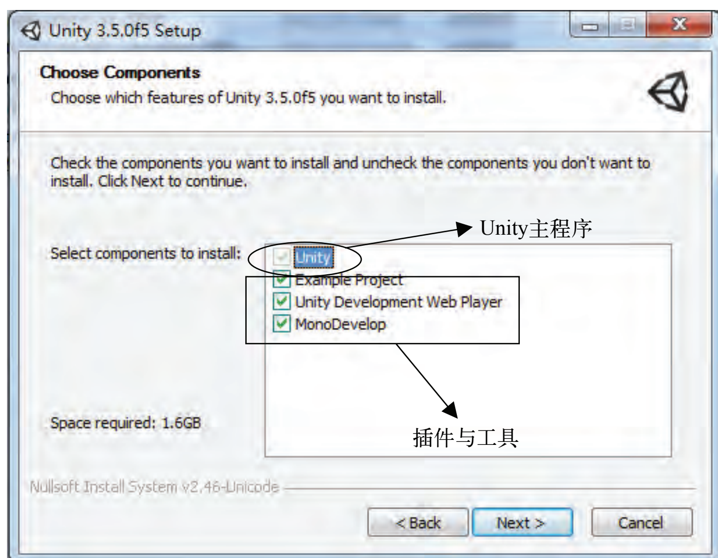


图1-8　选择安装界面

选择完插件与工具后，点击“Next”按钮，将打开确认安装界面，如图1-9所示，点击“Browse...”按钮，可设定Unity程序的安装路径。

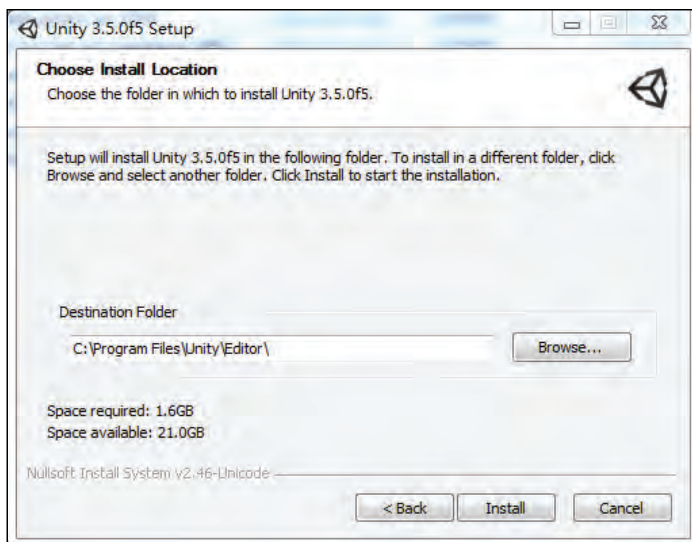


图1-9　确认安装界面

选择安装路径后，点击“Install”按钮，程序将开始自动安装。此时需要耐心等待一会儿，安装完毕后，程序将进入最后的安装完成界面，如图1-10所示，然后点击“Finish”按钮，将彻

底完成Unity的安装。如果在界面中勾选“Run Unity 3.5.0f5”复选框，安装完毕后Unity将自动被打开。

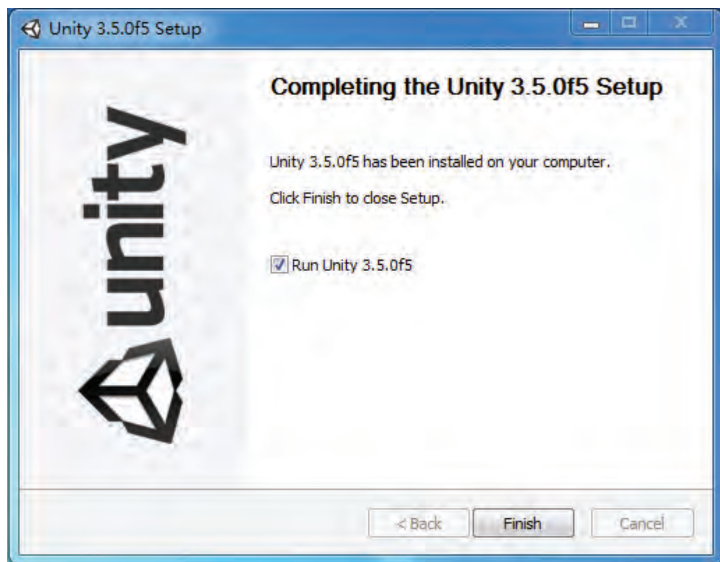


图1-10　完成安装

1.3　游戏界面对比

安装完Unity后，我们来打开它，首先映入我们眼帘的就是Unity的编辑器界面。在Mac OS 与Windows下，Unity的界面非常相似，并且它们之间的功能也都完全一样，包括制作游戏的方法、脚本的编写以及打包的过程等。它们之间唯一的一点小区别就是导航菜单栏的位置不太一样，但是导航菜单栏中的选项与功能都是完全一样的。所以说，无论在Mac OS中使用Unity还是在Windows中使用Unity，其开发过程完全一样。下面我们首先在Mac OS下打开Unity的界面。

1. Mac OS下的Unity界面

根据Mac OS操作系统自身的习惯，Unity的导航菜单栏位于屏幕顶部，如图1-11所示，其优点是它不会因为鼠标拖动下方引擎界面而发生位置的改变。导航菜单栏中包括Unity非常重要的一些功能，后面会向读者详细介绍其中的含义。



图1-11　导航菜单栏

首次进入Unity时会弹出欢迎窗口（如图1-12所示），如果不设置关闭，每次打开Unity时都会自动打开这个窗口，其有一些选项可以帮助我们学习Unity，下面介绍一些这些选项的具体含义。

- ❑ Video Tutorials：视频学习教程，这些都是Unity官方推荐的视频教程，它们非常全面，只可惜都是英文的。
- ❑ Unity Basics：使用事项，涵盖引擎自身的一些配置参数以及对电脑硬件的需求等。
- ❑ Unity Answers：问题与回答，读者可在这里与世界各地的朋友一起讨论Unity游戏开发。
- ❑ Unity Forum：Unity官方创建的开发者论坛。
- ❑ Unity Asset Store：资源商店，这里聚集着很多游戏开发所需的资源，有免费的也有收费的。



图1-12　Mac OS下的Unity界面

2. Windows下的Unity界面

图1-13为Windows下进入Unity的主界面效果图。和Mac OS下基本相同，只是界面的颜色与导航菜单栏的位置有点小小的区别，在Windows下Unity导航菜单栏可随窗口移动，而在Mac OS中，它则位于屏幕顶部。



图1-13 Windows下的Unity界面

1.4 购买许可证

Unity是一款收费的游戏引擎，读者可登录官网查看Unity许可证的购买地址与方式，官网访问地址为https://store.unity3d.com/shop/。

目前，可使用欧元、美元和日元购买Unity的许可证。如图1-14所示，打开Unity的购买网址，在“Store”的子页面标题中选择“Products”页面，在下方Unity版本中选择一个需要购买的版本，左侧为普通版本，右侧为加强版本，然后在右侧的下拉列表中选择购买Unity的币种。



图1-14 选择购买版本

选择完购买版本后，可以继续购买Unity配置插件。插件可以更好地帮助Unity开发游戏，目前配置插件只包括移动开发的Android平台与iOS平台，分为普通插件与加强插件。Team License

为团队许可证，多台电脑可使用Team License同时进行开发。如图1-15所示，其中已经列出了详细的购买参数。

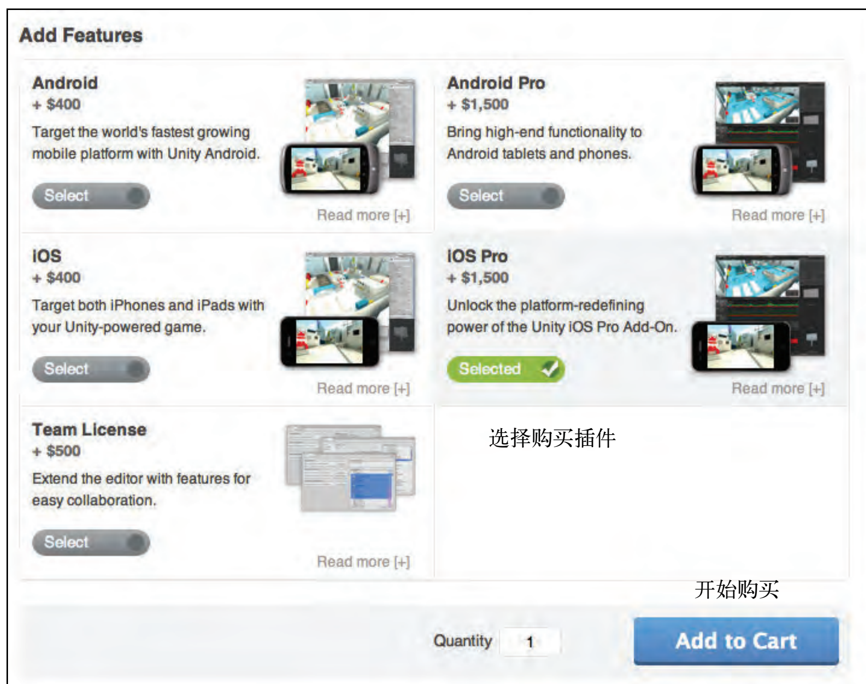


图1-15　选择配置插件

选择完合适的版本与插件后，点击右下角的“Add to Cart”按钮，将弹出确认购买页面，如图1-16所示，其中将出现之前选择购买的Unity版本与配置插件。确认无误后，点击“Check Out”按钮开始购买，系统将调出信用卡支付界面，根据提示即可完成购买。

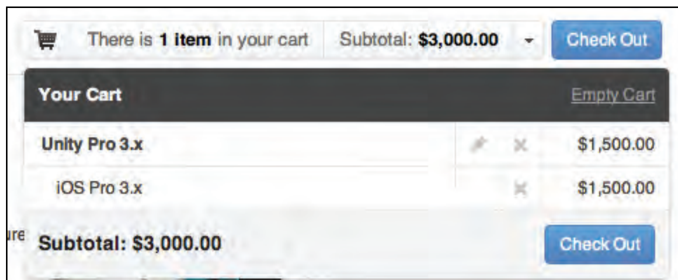


图1-16　确认购买页面

这里需要说明的是，购买时需要登录自己的Unity账号，如果之前没有注册账号，系统会提示你注册，或者自己登录官网注册，注册Unity的网址如下：https://store.unity3d.com/users/new。

如果已经购买过Unity许可证，比如之前购买的许可证是普通版本，既可以继续购买新版本，也可以为许可证进行加强版升级。不过升级也需要付款，支付金额与升级选项均与已有版本相关。如图1-17所示，在“Store”的子页面中选择“My Licenses”页面，登录Unity，然后就可以继续购买或升级自己的许可证。

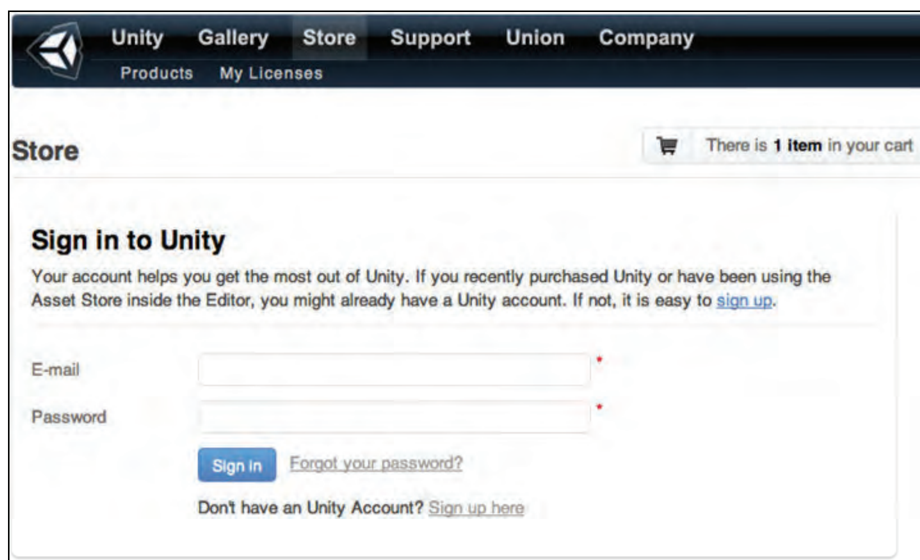


图1-17　升级

许可证购买成功后，会得到Unity提供的一串序列号，这串序列号需要在Unity程序中进行激活。在Unity导航菜单栏中选择“Unity”→“Enter Serial Number”菜单项，如图1-18所示，接着输入购买的许可证序列号即可完成激活。



图1-18　输入序列号

然后在导航菜单栏中选择“Unity”→“About Unity”菜单项，将弹出“About Unity”页面，如图1-19所示，许可证类型与序列号出现在界面的右下角。



图1-19　“About Unity”界面

1.5　打包与发布

首先我们需要找一个游戏工程来学习如何打包与运行游戏，在Unity官网中有很多可免费下载的示例程序，这是相当珍贵的学习资料，它们的下载地址为http://unity3d.com/support/resources/example-projects/。

下面我们介绍如何打包与发布游戏。首先选择一个比较完整的游戏示例来进行讲解，根据上述的下载地址将游戏工程“AngryBots”下载至本地。启动Unity，在导航菜单栏中选择“File”→“Open Project”菜单项，打开一个现有的游戏工程，这里我们选择打开刚才下载的“AngryBots”游戏工程。

默认情况下，打开游戏工程后，场景视图与游戏视图中是不存在任何游戏资源的，需要打开当前游戏对应的某个场景文件。如图1-20所示，我们打开“AngryBots”这个场景文件，此时场景视图与游戏视图中出现了该场景文件中游戏的所有资源。不同场景对应的游戏资源也会不一样，所以直接打开对应的场景文件即可。

如图1-20所示，在Unity界面右上角的下拉列表用于设置Unity界面的整体布局，默认布局为“Wide”，我习惯使用“2 by 3”布局，读者也可根据自己的喜好选择适当的界面布局。

游戏制作完毕后，需要进行平台打包才能最终发布。由于我已经购买了iOS平台的Unity许可证，所以本节将以iOS平台打包为例向读者介绍打包过程。

首先在Unity导航菜单栏中选择“File”→“Build Settings”菜单项，打开“Build Settings”窗口，如图1-21所示，在打包平台中选择iOS，然后点击右下角的“Build And Run”按钮，此时Unity将帮我们自动生成对应iOS平台的Xcode游戏工程。其他平台的游戏包制作方法与iOS平台完全一样。只需选择对应的打包平台，Unity遍可轻松实现跨平台游戏打包与发布。

注意　Xcode是苹果公司免费向开发人员提供的集成开发环境，用于开发Mac OS X应用程序。Xcode从3.1开始附带iOS SDK。

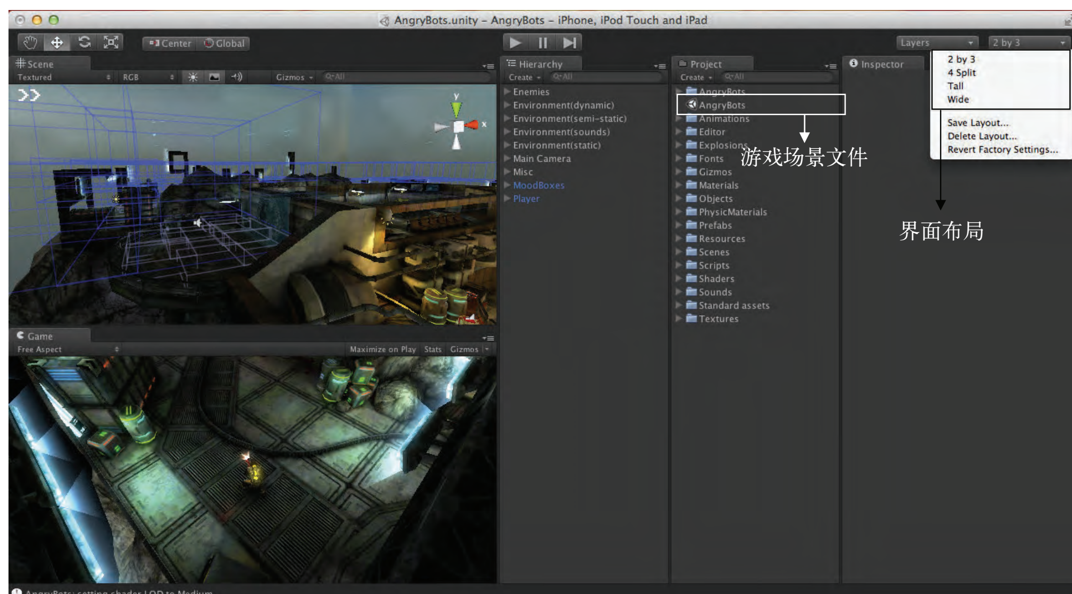


图1-20　界面布局



图1-21　“Build Settings”窗口

构建完iOS工程后，Unity会自动生成对应的Xcode游戏工程，其中包括运行在iOS平台下的所有Objective-C代码。使用Xcode打开Unity生成的游戏工程后，点击运行游戏按钮，构建的游戏工程与运行效果图将出现在我们面前，如图1-22所示。



图1-22 运行效果

在Windows下打包与运行的方式与Mac下完全一样，同样是在Unity导航菜单栏中选择“File”→“Build Settings”菜单项，此时打开Build Settings对话框，如图1-23所示，在打包游戏平台中选择Unity 3.5版本最新支持的Flash Player格式（目前免费），因为Flash Player格式需要Java虚拟机的支持，所以在构建项目之前请确保电脑中配置了Java环境。确保无误后，点击右下角的“Build and Run”按钮，即可在Windows下创建自己的项目。

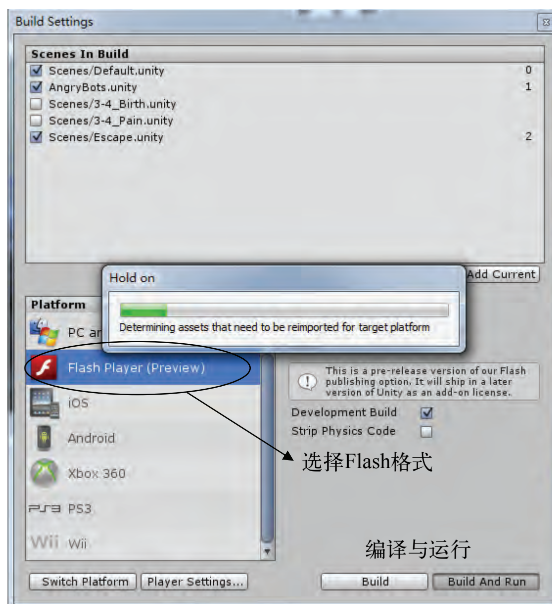


图1-23 在Windows下打包与运行

项目构建完毕后，Unity会将Flash文件生成在指定路径当中。找到生成的Flash文件，打开它即可在Windows下运行该Flash游戏，如图1-24所示。怎么样？跨平台开发很酷吧。

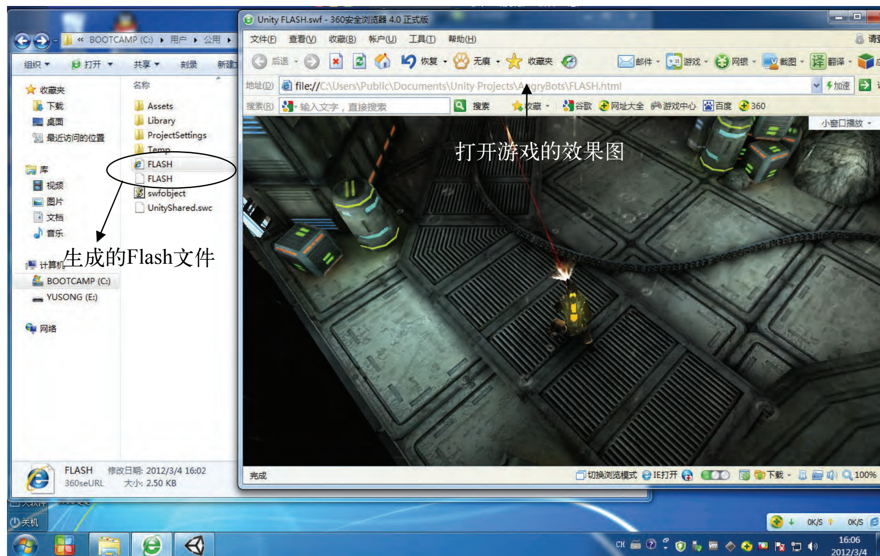


图1-24　打包与运行

上面我们介绍了Unity在iOS平台下的打包过程。Unity在iOS平台下打包后，会将整个Xcode工程提供给开发者，而其他平台不会像iOS那样将源代码提供给开发者，而仅提供一个可运行的文件，比如在Android平台中打包后将生成一个.apk包，PC and Mac平台下打包后生成对应的.exe与mac应用程序，Flash Player平台下打包后生成.swf格式的运行程序，WebPlayer平台下打包后生成网页格式的运行程序。

实际开发中可以通过消息传递或者插件的形式将游戏平台的相关代码加入到Unity工程中。例如，在Android平台下，Unity中无法使用Android系统提供的高级控件，可以将Android下的Java代码以插件的形式放在Unity工程中，最终通过传递消息的形式通知插件调用Android高级控件的方法。

1.6　本章小结

本章主要向读者介绍了学习Unity的基础知识，开发Unity 3D游戏之前的一切准备工作。首先介绍了Unity这款商业游戏引擎的特点，之后分别以Mac OS和Windows平台为例，简明扼要地介绍了这两种操作系统下Unity引擎的环境搭建以及它们之间的一些区别。然后介绍了Unity这款商业引擎的购买方式，学习了如何购买Unity及其相关配置插件。最后介绍Unity跨平台打包与运行的过程，以Mac OS下iOS平台与Windows下Flash平台为例向读者展示了完整的打包与制作方法。作为Unity学习的基础，希望读者们认真学习本章内容，为后续章节的学习做好铺垫。

第2章 编辑器的结构

Unity的编辑器界面非常友好，而传统游戏引擎几乎没有任何游戏界面，它们提供给开发者的往往都是些赤裸裸的源代码，以至于想实现任何功能，都需要编写代码才可以完成。而Unity理念是为开发者节省时间，让开发者爱上Unity并成为它的粉丝。它将大部分开发工作以可视化的方式提供给开发者，开发者无需编写实际代码，只需在界面中执行一些赋值操作就可以了。

2.1 游戏工程

Unity游戏工程由若干个游戏场景组成，在不同的场景中可实现不同的游戏效果，比如在菜单场景中可以实现游戏的所有菜单，而在游戏界面场景中可以实现游戏的所有业务逻辑等。游戏工程需要在Unity编辑器中打开，可视化的编辑器界面无疑能让开发者使用起来更加有条理，使开发者可以更清晰地看到整个游戏工程的层次与概念。使用Unity，我们可以快速敏捷地制作3D游戏。

2.1.1 创建工程

启动Unity后，界面最顶层的横向菜单栏为Unity导航菜单栏，所有工具类菜单项都在其中。

在导航菜单栏中选择“File”→“New Project”菜单项（如图2-1所示），程序将弹出“Project Wizard”窗口（如图2-2所示），在窗口上部选择“Create new Project”标签页。在“Project Directory”中点击“Set...”按钮设置游戏工程保存路径与项目名称，然后在“Import the following packages”中引入系统提供的标准资源包。Unity自身提供了一些资源包供开发者使用，其中包括一些常用脚本、贴图、特效、地形资源等。创建工程的时候，可以勾选所需引入的资源包，不过也可以在工程创建完毕后，在Project视图中动态地引入或修改这些资源包。确保无误后，点击界面右下角的“Create Project”按钮，完成新工程的创建。

游戏工程创建完毕后，Unity编辑器会自动打开这个工程，如图2-3所示，此时Unity编辑器的结构将立刻呈现在我们的眼前，从中可以清晰地看到Unity中包含的5大视图，它们十分有条理地分布在编辑器当中，从左到右的视图依次是：Scene（场景）视图、Game（游戏）视图、Hierarchy（层次）视图、Project（项目）视图和Inspector（监测）视图。视图与视图之间保持非常紧密的联系，为了使读者快速上手Unity这款游戏引擎，我们将按照一个有条理的顺序讲解这5大视图。首先介绍Project视图，这里主要存放当前游戏中的所有资源文件，比如游戏的贴图、动画模型以及

一些声音文件等。然后介绍Hierarchy视图，这里可以创建一些模型，比如立方体、球体、平面等，但是这些模型只具备单一的3D网格，如果要给它们添加资源，比如添加一些图片纹理，就需要在Project视图中将贴图文件赋值给Hierarchy视图中的模型本身。接着讨论一个特殊的视图，那就是Inspector视图，它主要用来呈现某个游戏对象或者游戏组件的一些说明与参数信息。接下来讨论Scene视图，Hierarchy视图中创建的模型都会出现在Scene视图当中，在Scene视图中可以修改模型的位置、旋转的角度和缩放的大小等。可见，Scene视图是用来编辑整个游戏世界的。最后是Game视图，Game视图是游戏最终发布后展示在屏幕中的效果，屏幕展示的内容为Hierarchy视图当中摄像机照射的部分，所以游戏发布时必须确保Hierarchy视图中具有摄像机对象，否则Game视图将一片漆黑。



图2-1　导航菜单栏

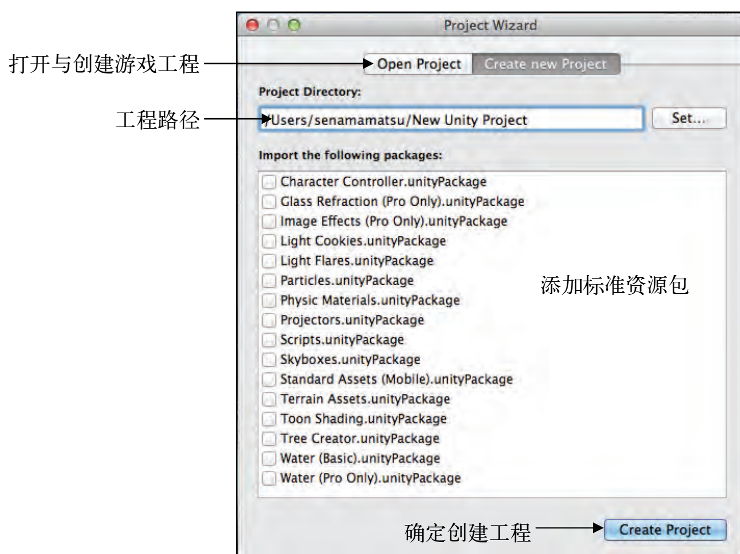


图2-2　“Project Wizard”窗口

乍看起来，Unity编辑器中的选项非常繁杂，读者可能会想Unity会不会不好学啊？请读者不要为此担忧。随着后面的学习，你会逐渐爱上Unity编辑器，因为它的界面友好，功能又相当强大，最为重要的是它是一款非常容易上手的游戏引擎。

小技巧 将鼠标放置在任意视图中，然后按下空格键即可最大化该视图，再次按下空格键将还原视图。



图2-3 Unity编辑器

2.1.2 打开工程

打开Unity游戏工程的方法有两种，其中最简单的方式就是在“我的电脑”中找到游戏工程的场景文件，双击这个场景文件即可。场景文件以.unity为后缀，比如“AngryBots.unity”文件就是该游戏中的一个场景文件，如图2-4所示。一般情况下，整个游戏工程中包含多个游戏场景文件，打开任意场景文件都可以打开游戏工程及其对应的场景。

另外一种打开游戏工程的方式是在Unity导航菜单栏中选择“File”→“Open Project”菜单项，此时系统将弹出“Choose Project Directory”界面（如图2-5所示），在“我的电脑”中选择将打开游戏项目的根目录文件，然后点击界面下方的“Open”按钮即可。

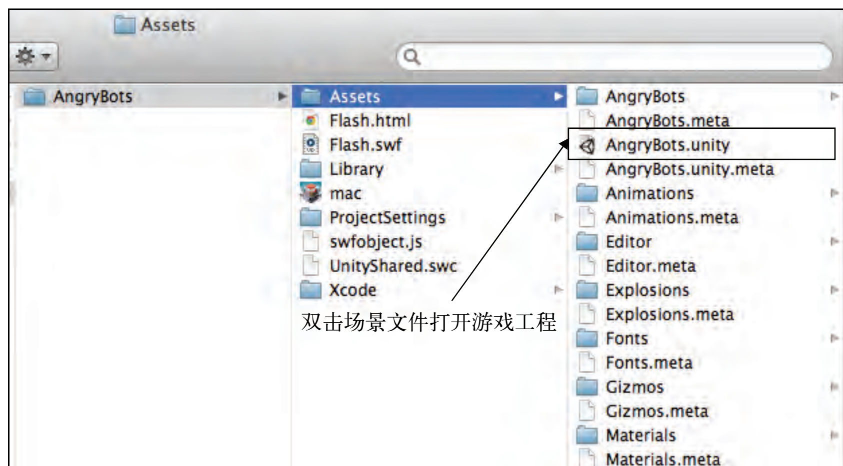


图2-4　场景文件

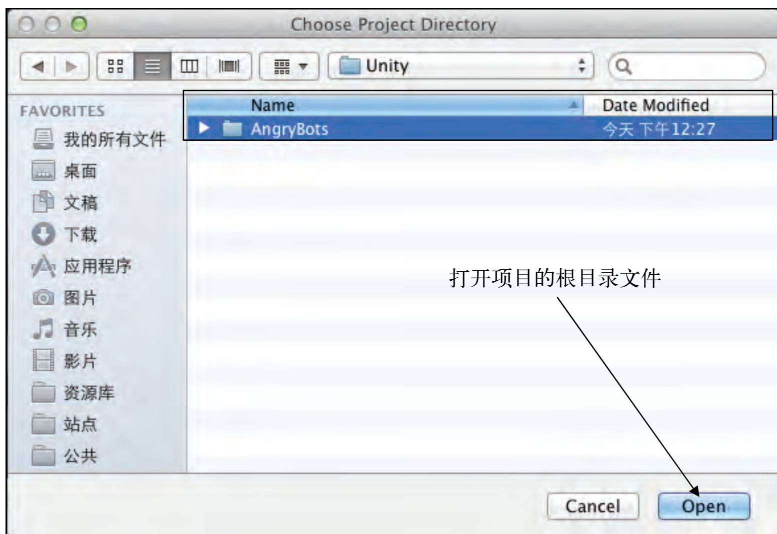


图2-5　打开工程

2.2　Project 视图

Project视图主要存放游戏中用到的所有资源文件，常见的资源包括：游戏脚本、预设、材质、动画、自定义字体、纹理、物理材质和GUI皮肤等，这些资源需要赋予Hierarchy视图中的某些游戏对象。在Hierarchy视图的左上角点击“Create”按钮，将弹出一个下拉列表，如图2-6所示。通过这个下拉列表，可以创建游戏的相关资源。下面简要介绍一下这个下拉列表中各个选项的含义。

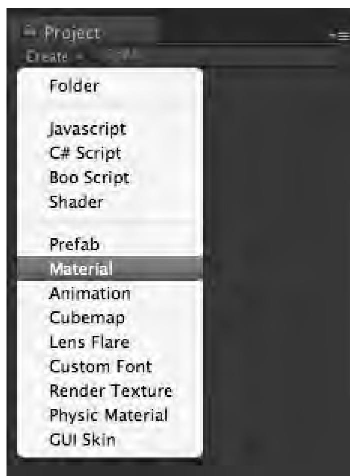


图2-6 下拉列表

- Folder：文件夹，用于资源的分类。
- Javascript：JavaScript脚本文件。
- C# Script：C#脚本文件。
- Boo Script：Boo脚本文件。
- Shader：设置一个着色器，可以用ShaderLab编写着色器代码。将着色器绑定在材质身上，可直接影响材质的显示效果。
- Prefab：预设，用于场景中游戏对象的克隆。使用预设，可以有效避免过多重复的游戏对象占用内存的情况，后续会详细介绍。
- Material：材质，用于为模型添加颜色与贴图。
- Animation：游戏动画。
- Cubemap：创建具有六个面的贴图资源，用于立方体或天空盒子的贴图。
- Lens Flare：添加镜头光晕效果。
- Custom Font：自定义字体。
- Render Texture：渲染贴图。
- Physic Material：物理材质，用于调整对象的物理属性，比如摩擦力和弹力等。
- GUI Skin：图形用户界面，可以为多个控件添加样式。

创建完资源后，它将保存在工程根目录下的“Assets”文件夹中。另外，也可以使用鼠标将现有的资源从外部拖曳至Project视图中，或者直接将资源文件拖入项目根目录下的“Assets”文件夹中，然后在Unity编辑器中刷新一下，即可在Project视图中看见刚刚托入的游戏资源。刷新方法是：将鼠标放在Project视图中，然后单击鼠标右键，选择“Refresh”菜单项即可。在Project视图中选择任意游戏资源，用鼠标右键点击“Reveal in Finder”菜单项（在PC中是“Show in Explorer”），即可在“我的电脑”中找到对应的资源。

下面我们通过一个具体的示例简要介绍一下如何创建一个常用的资源文件。首先在Project视图中点击“Create”→“Material”菜单项，创建一个普通的游戏材质。游戏材质为游戏对象的显示内容，它可以设置颜色或贴图。将材质赋予游戏对象时，游戏对象将显示这个材质，如图2-7所示。材质创建完毕后，可在右侧的Inspector视图中看到所有相关信息。Shader下拉列表显示材质的着色器，用于设置材质的渲染方式，如透明和混合等，默认着色器的渲染方式为Diffuse。点击Main Color右侧的图标，将弹出颜色选择窗口（如图2-8所示），选择任意颜色后，“H”、“S”和“V”对应的颜色值会发生改变。这里需要说明的是，“A”比较特殊，它表示材质的透明度，0为完全透明，255为完全不透明。

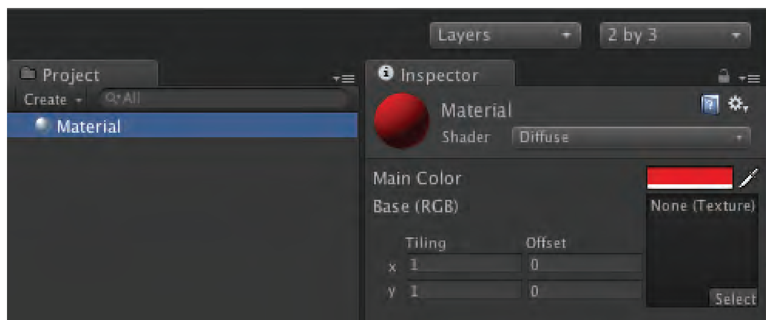


图2-7　创建材质

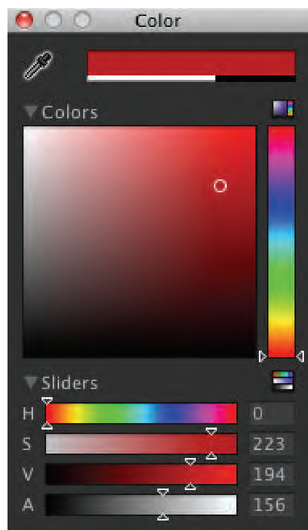


图2-8　设置颜色界面

使用材质，不仅可以设置颜色，还可以设置贴图，在“None（Texture）”中点击“Select”按钮，可在Project视图中选择贴图文件。此外，同一个材质还可以同时设置贴图与颜色。

2.3　Hierarchy 视图

Hierarchy视图主要存放游戏场景中具体的游戏对象，比如摄像机、平面贴图、3D贴图、光源、箱子、球体、胶囊体、模型、平面和地形等。任何一个全新的游戏工程创建完毕后，默认都会创建一个游戏场景并且将主摄像机添加在该场景的Hierarchy视图中。对于3D游戏来说，摄像机可以让我们以不同的角度观察游戏世界。

我们知道任何一款游戏都由若干个场景构成，而任何一个场景都由若干个对象构成。Hierarchy视图中的游戏对象与Project视图中的游戏资源密切相关，就好比任何一个漂亮的东西都需要美丽的外表来衬托。比如在Hierarchy视图中创建一个立方体，在Project视图中存放这个箱子的显示材质，如果将材质赋予立方体对象，那么立方体对象就好比穿上了衣服，它的表面将发生改变。

下面我们将学习如何创建游戏对象。在Hierarchy视图中点击左上角的“Create”按钮，此时将弹出一个下拉列表，从中选择一个需要创建的游戏对象即可，如图2-9所示。此外，部分游戏模型对象也可从Project视图的资源中拖曳过来，比如游戏的模型资源，它既可以说是游戏资源也可以说是游戏对象，使用时需要将它拖曳至Hierarchy视图中，但是模型的贴图资源文件则只能在Project视图当中。游戏对象创建完毕后，即可在Scene视图中看到我们创建的游戏对象。

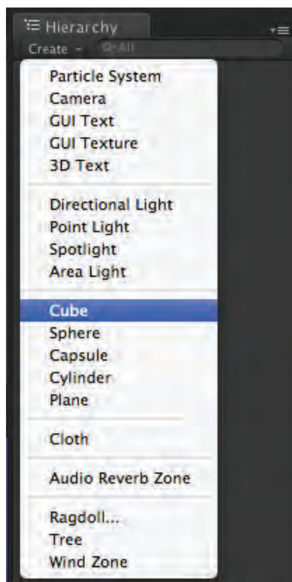


图2-9　创建对象

在Hierarchy视图中可创建的游戏对象繁多，下面我们简要介绍一下图2-9中各选项的含义。

- Particle System：粒子效果发射器，可处理游戏中的所有粒子效果。
- Camera：游戏摄像机。

- GUI Text：3D平面文字。
- GUI Texture：平面贴图。
- 3D Text：3D立体文字。
- Directional Light：定向光源，常用于天空向地面定向投射的光源。
- Point Light：点光源，在一定范围内照亮的光源。
- Spotlight：聚光灯，与点电源类似，但光线只照亮一个圆锥区域。
- Area Light：区域灯，可照亮整个选定的区域。
- Cube：立方体。
- Sphere：球体。
- Capsule：胶囊体。
- Cylinder：圆柱体。
- Plane：平面。
- Cloth：布料。
- Audio Reverb Zone：音频效果。
- Ragdoll：布娃娃效果。
- Tree：树模型。
- Wind Zone：风向，可影响游戏中的对象。

2.4 Inspector 视图

Inspector视图相对来说比较复杂，读者暂时可以把它理解成存放游戏对象、游戏资源、游戏设置以及展示描述信息的地方。无论是在Project视图中选择一个游戏对象，还是在Hierarchy视图选中一个游戏资源，或是在引擎中选择任意一个控件时，Inspector视图都会被打开，它将展示选择对象的所有描述信息。在该视图中，会详细列出选择组件的描述以及该组件描述的所有参数，并且部分组件参数是可动态修改的。修改完Inspector视图的参数后，在Game视图中可直接看到修改后的效果。

2.4.1 简介

在本节中，我们将创建一个立方体游戏对象，并且在Inspector视图中查看其参数的信息。在Hierarchy视图中，点击“Create”→“Cube”菜单项，创建一个立方体对象。选择该立方体对象，此时右边的Inspector视图便被打开，立方体对象相关的描述信息均显示在其中，如图2-10所示。

另外，在Inspector视图显示的每一个组件中，都可以通过点击该组件右上角那个小问号去查阅具体信息。点击小问号后，将直接链接到官方的用户手册中，其中详细介绍了相关组件。

如图2-10所示，该立方体对象在Inspector视图中包含的信息如下所示。



图2-10 Inspector视图

- ❑ Transform：模型的变化，通过它可动态修改立方体的三维坐标。
  - ■ Position：该立方体的位置。
  - ■ Rotation：该立方体的旋转角度。
  - ■ Scale：该立方体的缩放比例。
- ❑ Cube（Mesh Filter）：网格过滤器的类型，它可以直接确定该模型的物理材质，这里默认为Cube。
- ❑ Box Collider：立方体碰撞器，它与刚体紧密结合。在介绍Unity物理引擎相关的知识中，我们会详细介绍它。
- ❑ Mesh Render：网格的绘制，它可对网格进行材质的渲染。
  - ■ Cast Shadows：网格是否投射阴影。
  - ■ Receive Shadows：网格是否接收阴影。
- ❑ Materials：设置材质的资源。

2.4.2 平台设定

在游戏平台设定中，可设置游戏的一些平台属性，比如游戏图标、Logo和游戏名称等。首先在Unity导航菜单栏中选择“Edit”→“Project Settings”→“Player”菜单项，此时编辑器将弹出平台设定窗口（如图2-11所示），在Inspector视图中可进行游戏平台的相关设置。

在PlayerSettings（玩家设置）中，可设定开发公司的名称、程序名称和默认程序图标等。在Per-Platform Settings（平台设定）中，可选择待打包的游戏平台，比如Web平台、PC/Mac平台、iOS平台以及Unity 3.5最新发布的Flash平台。各平台之间的设定都大同小异，设定完毕后，打包

游戏后即可看到设定的效果。

下面简要介绍一下平台中4大设定的含义。

- ❑ Resolution and Presentation：屏幕的尺寸以及位置等。
- ❑ Icon：程序的图标。
- ❑ Splash Image：开机预览图。
- ❑ Other Settings：其他设定，这里主要设置一些平台的相关特性。



图2-11　平台设定窗口

设定完平台后，就可以直接打包编译工程了。在编译的时候，会根据对应平台而选择平台设定中的信息。

实际上，Inspector视图中包含的知识非常多，不过原理都大同小异。选中任何一个模型或者资源后，描述信息都会出现在Inspector视图中。对于读者来说，一时很难彻底理解其中的含义，不过不要紧，本章中读者只需要知道Inspector视图中大致存放着什么东西即可。在后面的章节中，我们会结合实际项目带领大家由浅入深地学习Inspector视图的知识。

2.5　Scene 视图

Scene视图主要存放游戏中的模型资源。场景中3D模型的种类繁多，比如游戏主角、敌人、

NPC 、道具、天空、山川、河流和云彩等，这些游戏模型的对象都存储在Hierarchy视图中，而与这些模型相关的贴图资源文件则又保存在Project视图中。

下面还是以之前创建的立方体对象为例进行介绍。首先在Hierarchy视图中选择立方体对象，然后在Scene视图中按快捷键“F”来近距离查看该游戏对象。如图2-12所示，可以发现创建的立方体周围含有3个箭头，它们的颜色分别是：“红”、“绿”和“蓝”。3个箭头代表3个坐标方向，红色为x轴方向，绿色为y轴方向，蓝色为z轴方向。使用鼠标拖动任意一个箭头所指向的方向来拖动模型，就可以修改该模型的3D坐标。

2.5.1　视图介绍

在Hierarchy视图中创建的所有游戏对象都可以保存在同一个Scene视图中，而Scene视图中的游戏对象将直接影响游戏发布的效果。Scene视图可以非常庞大，但是游戏发布后在屏幕中显示的内容只可能是摄像机照射的一部分，所以动态移动摄像机的位置即可刷新屏幕显示区域。

创建游戏对象后，默认的材质是灰色的。首先创建一个立方体对象，按照图2-12箭头指示的方向，将创建的红色游戏材质拖曳至立方体后面，即可完成资源到对象的赋予，此时Scene视图中立方体对象的颜色立刻发生改变，变成了红色。

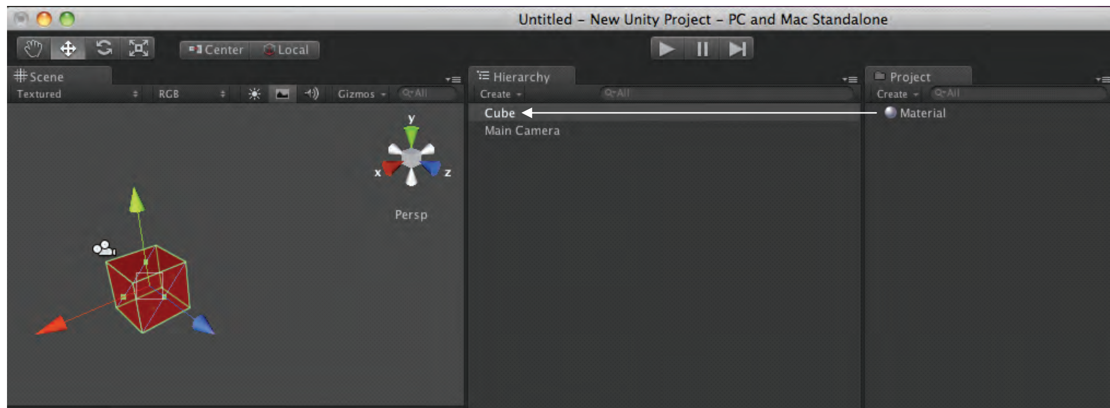


图2-12　Scene视图

另外，主摄像机（Main Camera）的位置也是非常重要的属性。主摄像机投射的部分就是游戏发布后屏幕中显示的内容。如果Scene视图中确定存在模型，但是运行游戏后发现Game视图中没有显示任何内容，那么肯定是主摄像机没有射向Scene视图中的模型，此时修改一下Main Camera对象的位置即可。

此外，在Hierarchy视图中选择摄像机后，在Scene视图右下角会出现游戏预览视图Camera Preview，如图2-13所示，从中可清楚地看到摄像机目前所照射的部分。使用Camera Preview，开发者就能在编辑场景时方便地调节场景中模型的位置。如果不小心将Main Camera删掉的话，只须在Hierarchy视图中使用“Create”→“Camera”菜单项重新创建一个主摄像机对象即可。



图2-13 预览视图

2.5.2 移动视图

Scene视图是以2D平面的形式展现出来的，但是场景中却是3D游戏世界，因此要想在2D平面中清晰地观察模型在3D世界中的所有位置，确实有点难度，但是可以通过旋转或者缩放等方式移动Scene视角去观看场景中的模型。Unity编辑器提供了一组移动视图的快捷键，方便开发者动态观察模型。

移动视图前，首先要确保鼠标指针停留在Scene视图当中，然后即可通过快捷键进行旋转、缩放和平移等操作。快捷键的使用方式如下所示。

- 旋转视图：使用Option + 鼠标左键（在PC键盘上，则是Alt键 + 鼠标左键）可以任意拖动鼠标来旋转视图。
- 缩放视图： 滚动鼠标中键可以缩放整体视图。
- 平移模型： 使用鼠标左键可以任意平移模型。

在Scene视图上方有一组处理模型变换的工具栏（如图2-14所示），它一共由4个工具按钮组成，从左到右依次为：“拖动工具”按钮、“移动工具”按钮、“旋转工具”按钮和“缩放工具”按钮。下面先来简单介绍这4个工具按钮的含义。

- 拖动工具：整体拖动Scene视图。
- 移动工具：移动模型。
- 旋转工具：旋转模型。
- 缩放工具：缩放模型。



图2-14 变换工具栏

1. 移动模型

在Scene视图中选择需要移动的模型，然后在变换工具栏中点击“移动工具”按钮（快捷键为 W），此时在Scene视图中将出现移动模型的3个箭头，如图2-15所示。使用鼠标按照箭头所指的方向拖动模型，即可整体修改该模型在3D世界中的坐标。在模型移动的过程当中，可以在Inspector视图中看到具体模型移动后的坐标值。

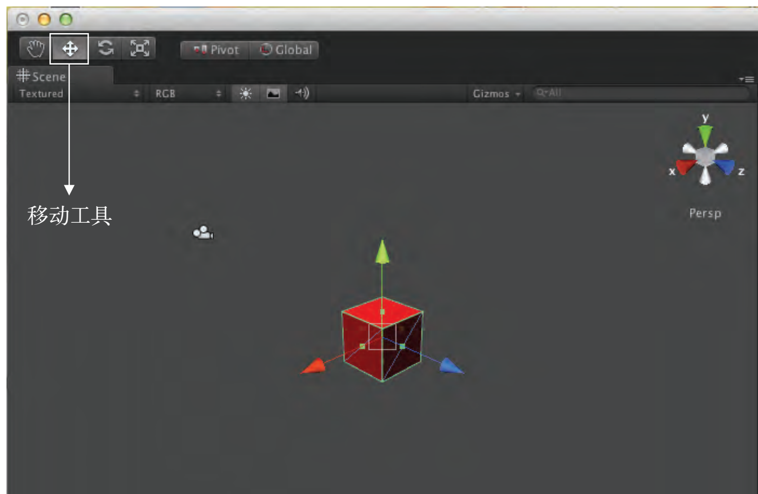


图2-15　移动模型

下面我们来学习一条简单的移动脚本。首先在Project视图中选择“Create”→“JavaScript”菜单项，创建一条游戏脚本。打开这条游戏脚本，将以下代码写入脚本当中：

```
function Update()
{
    //移动模型
    transform.Translate(Vector3.forward * Time.deltaTime);
}
```

这里的`Update()`方法为系统所调用，系统在每一帧都会调用该方法。该方法前面的`function`关键字为JavaScript语言的标准用法，表示后面跟随的`Update()`是一个方法。

`transform.Translate()`方法用于设置游戏对象平移的方向，该方法中的参数为移动的方向，其中`Vector3.forward`表示移动的方向为前方，`Time.deltaTime`表示`Update()`方法上一帧持续的时间，它俩的乘积就是沿着前方模型一次移动的距离。在本示例中，运行游戏后，立方体将缓慢向前移动。

代码写入完毕后，直接将这条JavaScript脚本绑定在立方体中，具体的绑定方法很简单：直接把这个脚本对象拖曳至Hierarchy视图中的游戏对象即可。

2. 旋转模型

在Scene视图中，可以按照任意角度旋转游戏对象。在Scene视图中，选择需要旋转的模型并

且在变换工具栏中选择“旋转工具”按钮（快捷键为 E），此时该游戏对象的周围将出现以不同颜色标注的3个旋转轨迹（如图2-16所示），其中红色轨迹代表沿x轴旋转，绿色轨迹代表沿y轴旋转，蓝色轨迹代表沿z轴旋转，可按3个旋转轨迹拖动鼠标来整体旋转该模型，并且在旋转模型的过程当中，在右侧的Inspector视图中即可看到旋转时的数值。此外，也可直接输入具体数值来修改模型的旋转系数。

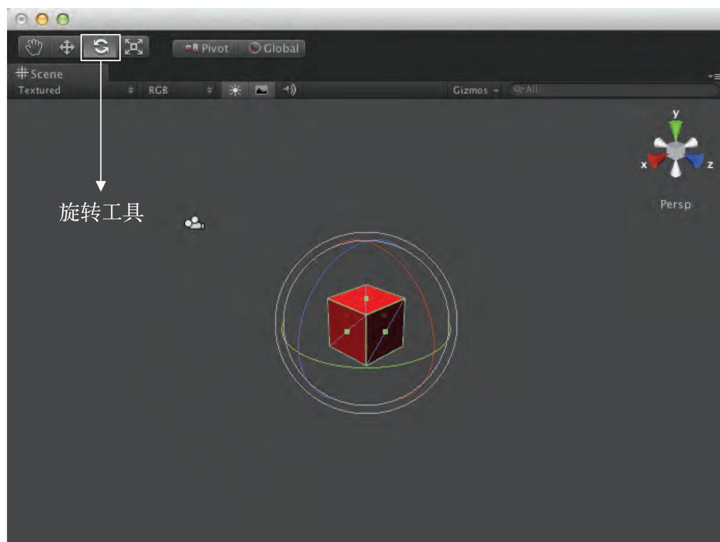


图2-16　旋转模型

下面我们来学习一条简单的旋转脚本，具体如下所示：

```
function Update()
{
    //旋转模型
    transform.Rotate(Vector3.up *Time.deltaTime);
}
```

在上述代码中，`transform.Rotate()`方法用于绕自身旋转模型，其参数分别是旋转的角度与方向，`Vector3.up`表示模型自身旋转的方向为y轴，`Time.deltaTime`表示上一帧所持续的时间，它们的乘积表示模型一帧内旋转的角度，最后将这条脚本绑定在立方体中。因为系统每帧都会调用`Update()`方法，所以运行游戏后，立方体将缓慢地绕着y轴自身旋转。

3. 缩放模型

通过上面的学习，相信大家能猜到缩放模型的方式。在变换工具栏中选中“缩放工具”按钮（快捷键为 R），在Scene视图中选择需要缩放的模型，拖动不同的小方块即可缩放该模型，如图2-17所示。在图2-17中，一共含有4个小方块，它们代表着不同的含义，红色小方块代表沿x轴缩放，绿色小方块代表沿y轴缩放，蓝色小方块代表沿z轴缩放，黄色小方块代表缩放整个模型。此外，向上拖动鼠标指针为放大模型，向下拖动鼠标指针为缩小模型。此外，模型缩放的系数同样

可在右侧的Inspector视图中查看与修改。

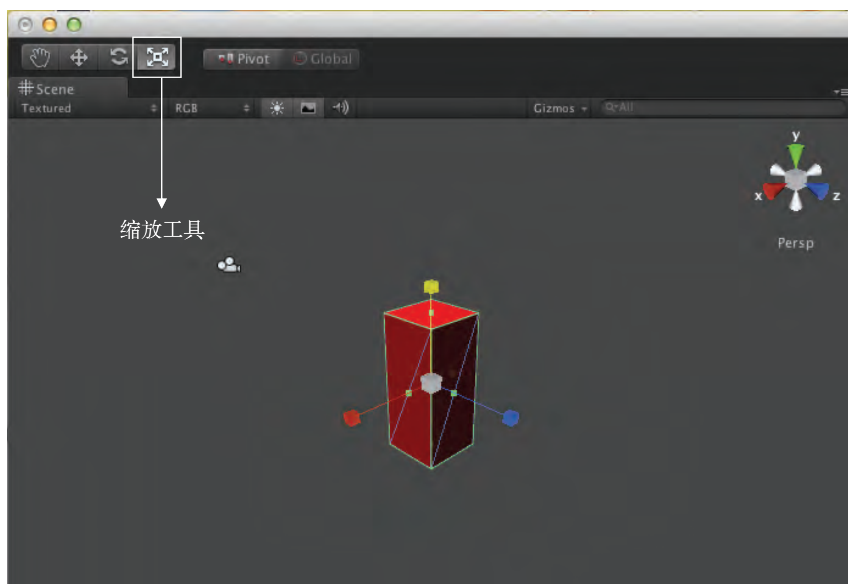


图2-17　缩放模型

现在我们已经学会如何移动、旋转和缩放模型了。其实无论是移动模型、旋转模型还是缩放模型，都是在改变模型在3D世界中的坐标。在Scene视图中选择任意模型后，右侧的Inspector视图会记录游戏场景中模型的位置系数、旋转系数和缩放系数。如图2-18所示，在下面的方框内可以清晰地看到这3组数值，它们的含义如下所示。

- Position：模型的位置。
- Rotation：模型的旋转系数。
- Scale：模型的缩放系数。

3组数值对应的X、Y和Z数值框记录着这个模型在3D世界中的变换。此外，这些数值可以直接手动修改，然后无需运行游戏即可在Scene视图中看到模型修改后的效果。

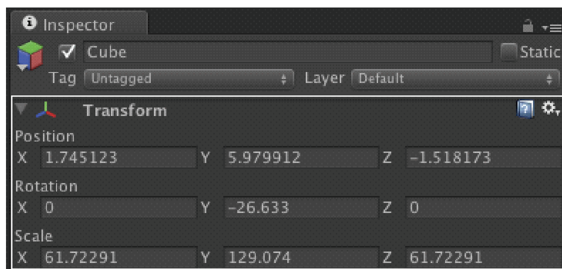


图2-18　变换数值

2.5.3　场景工具

使用场景工具（Scene Gizmo），可方便查看Scene视图中的游戏对象。场景工具位于Scene视图中的右上角，如图2-19所示。可以看出，它由x、y、z这3个彩色的小锥形和3个灰色的小锥形组成，中间的灰色小方块将它们连接在了一起。

点击任意一个彩色小锥形，可改变游戏场景的整体视角：点击“x”轴红色小锥形，视角将以该模型的x轴方向呈现；点击“y”轴绿色小锥形，视角将以该模型的y轴方向呈现；点击“z”轴蓝色小锥形，视角将以该模型的z轴方向呈现。点击中间的灰色小方框，视角将还原为以默认的45° 角方向呈现。

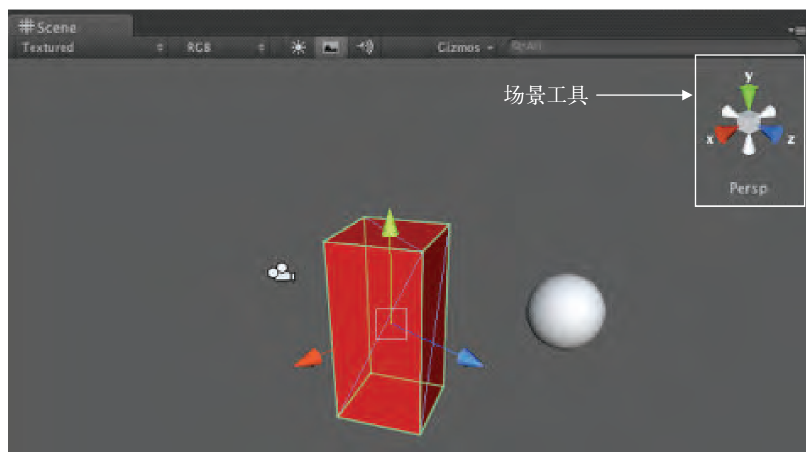


图2-19　场景工具

该场景工具非常适合查看游戏对象视角，为什么这么说呢？在Scene视图中，模型以2D平面的形式展现出来。因为2D平面只能确定两个轴的方向与距离，所以在编辑场景时很难在一个平面中确定两个模型之间的三维距离，并且在2D平面中观察3D模型必然会出现误差。而使用场景工具，就能轻松避免该问题。

一旦选择视角方向后，就可以清楚地看到某个轴方向所对应的平面。为了使读者更清楚地感觉到该场景工具存在的意义，下面我们制作一个测试示例。如图2-19所示，在红色立方体旁边创建一个小球作为参照物，然后我们观察这两个模型的位置。目前，我们发现小球好像紧挨着红色立方体。其实不然，我们的眼睛被2D平面欺骗了，为什么呢？在场景工具中点击“y”轴绿色小方块，将场景视角朝向修改为y轴。如图2-20所示，立方体和小球在三维坐标系中的x轴方向上相差的距离其实非常远，而在使用默认的45° 角去观察模型时，感觉它们离得很近。因此，在开发中（尤其是在编辑游戏场景时）善用场景工具来定位视角的x轴、y轴、z轴是至关重要的一步。

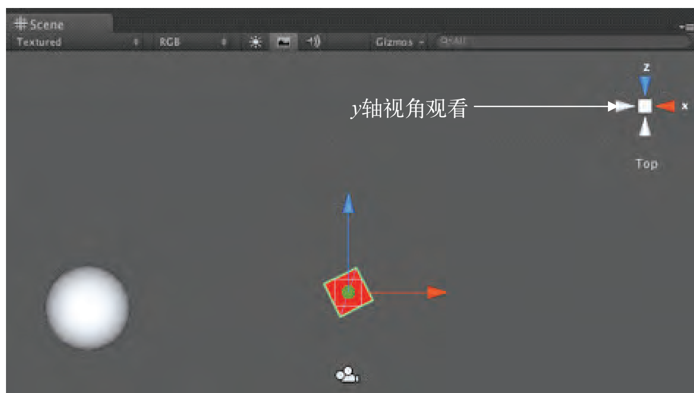


图2-20 y轴视角

2.5.4 Scene视图控制条

如图2-21所示，Scene视图控制条位于Scene视图的最上方，它是由多个下拉列表构成的一个控件。视图控制条主要的功能是切换Scene视图中显示的模式，比如设置绘图模式、渲染模式、灯光效果、游戏声音等。

修改显示模式后，可让开发者在Scene视图中更为清晰地看到模型的位置，因为如果游戏场景中模型的种类过多，会给开发者带来眼花缭乱的感觉，很难分辨其中的差异。



图2-21 Scene视图控制条

在Scene视图中设置视图显示模式后，不会影响到游戏的最终发布效果，它只用于在开发阶段方便开发者调试程序。下面简要介绍一下视图控制条中的绘图模式和渲染模式。

1. 绘图模式

绘图模式（DrawMode）可修改Scene视图中所有模型的绘制方式。在Scene视图控制条中，共有5种绘图模式，通过“Textured”下拉列表（如图2-22所示）可设置绘图模式。默认情况下，绘图模式为 Textured，意思是显示所有贴图。选择任意一个绘图模式后，可直接在Scene视图中看到效果。为方便大家学习，我们简要介绍一下这5种绘图模式的具体含义。

- Textured：完全显示所有模型的游戏贴图。
- Wireframe：只显示模型的网格线框。
- Tex-Wire：完全显示所有模型的游戏贴图以及所有模型的网格线框。
- Render Paths：设置渲染模式提前或者延迟。
- Lightmap Resolution：设置场景之上的栅格网。

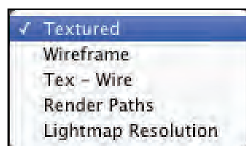


图2-22　绘图模式

下面举例说明一下。如图2-23所示，从“Textured”下拉列表中选择“Wireframe”贴图渲染模式，我们可直接看到模型的所有网格，并且过滤掉模型原有的贴图。

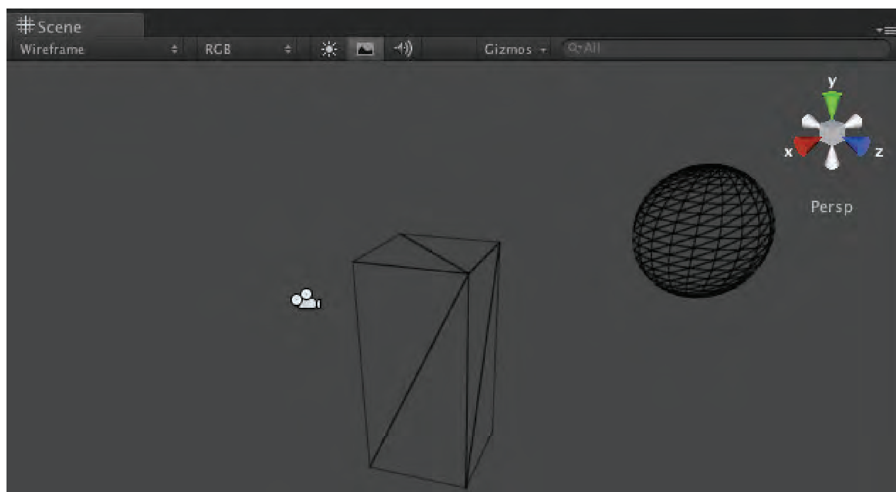


图2-23　Wireframe模式

2. 渲染模式

在Scene视图控制条中，可选择4种渲染模式（RenderMode）。点击“RGB”下拉列表（如图2-24所示），可设置颜色的渲染模式，默认渲染模式为RGB。选择任意一个绘图模式后，可直接在Scene视图中看到效果。为了方便大家学习，我们简要介绍一下这4种渲染模式的具体含义。

- RGB：默认渲染模式，正常的游戏渲染颜色。
- Alpha：阿尔法半透明渲染模式。
- Overdraw：设置透明模式。
- Mipmaps：设置理想的纹理尺寸。



图2-24　渲染模式

在渲染模式下拉列表右侧，还有3个按钮，分别代表是否开启场景照明灯光、是否显示场景网格、是否开启音乐。

2.6 Game 视图

在游戏场景中进行各种复杂的编辑后，终于到了最终展示游戏效果的地方，这就是Game视图。Game视图中显示的内容完全取决于摄像机所照射的部分。我们可以通过在Scene视图中移动摄像机的位置与旋转角度或者摄像机照射的范围，来修改Game视图中显示的内容。

运行游戏后，即可在Game视图中看到之前编辑的游戏效果。因为游戏发布后的主画面完全取决于Scene视图中主摄像机投射的区域，所以Scene视图中的摄像机一定要照射正确的位置。为了提升游戏的可玩性与显示效果，通常游戏开发中会存在多个摄像机，它们之间根据游戏剧情的需要进行相互切换，以各种角度去投射游戏场景。

2.6.1 运行游戏

在Unity编辑器上方还有3个重要的按钮（如图2-25所示），从左到右依次为运行游戏、暂停游戏和逐帧运行游戏，下面我们先来简单介绍一下这3个按钮的功能。

- “运行游戏”按钮：用于开始运行当前游戏。
- “暂停游戏”按钮：用于暂停正在运行的游戏，该按钮必须在游戏运行时点击才有效果。
- “逐帧运行游戏”按钮：主要用于程序的调试，每点击一次，游戏运行一帧。



图2-25 运行游戏

Unity支持在游戏运行过程中可以继续编辑游戏，比如如果游戏正在运行，那么在Scene视图中移动模型的位置时，在Game视图中可以直接看到模型发生了改变的效果。但是结束游戏的时候，所有数据又会还原成运行游戏之前的样子，这可能是Unity出于安全性的考虑，这一点大家需要注意一下。编辑游戏场景的时候，一定要确保游戏没有在运行当中，或者在游戏运行状态中将编辑后游戏的相关数值记录下来，然后结束游戏，最后再将编辑时记录的数值重新编辑在游戏当中。

2.6.2 Game视图控制条

Game视图控制条（如图2-26所示）位于Game视图的顶端，它用来控制Game视图中显示的一些属性，比如屏幕强制显示的宽高比例、最大化与最小化运行游戏之间的切换、显示当前游戏运行中的一些重要参数等，下面简要介绍一下相关的按钮控件。

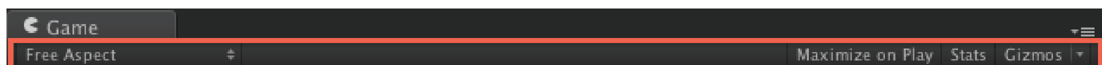


图2-26　Game视图控制条

“Free Aspect”下拉列表（如图2-27所示）用于设置游戏显示的宽高比例，默认的游戏显示比例为“Free Aspect”（完全填充）。此外，该下拉列表中还包含一些屏幕显示比例的选项，从中可以直接选择屏幕显示比例。一旦选好屏幕显示比例后，在Game视图中即可看到显示比例的效果。显示比例的调整不会影响游戏的最终发布，所以在开发过程中我们可以使用它方便地模拟不同分辨率的设备中显示的比例。

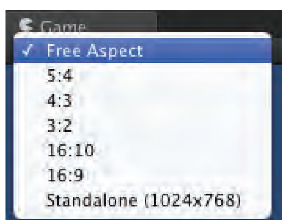


图2-27　“Free Aspect”下拉列表

“Maximize on Play”选项按钮用于最大化游戏屏幕。一旦按下该按钮，那么运行游戏时，Game视图将最大化屏幕，如果该按钮未按下，Game视图将以默认形式展现。如图2-28所示，当前Game视图中的游戏已经处于全屏运行当中，退出游戏后，Game视图还原为默认状态。

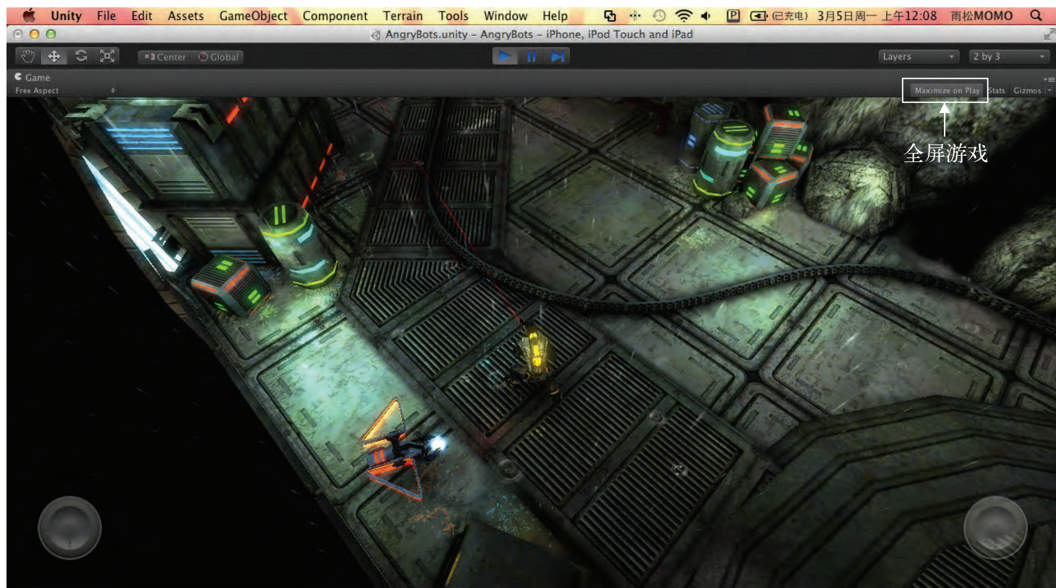


图2-28　最大化游戏

“Stats”选项按钮用于设置运行游戏时是否开启游戏的状态窗口。如图2-29所示，点击“Stats”按钮时，运行游戏后，Game视图将显示出所有图形渲染的参数以及网络连接的具体参数，这些参数是程序调试时依据的一些重要数据。再次点击“Stats”按钮，即可关闭状态窗口。

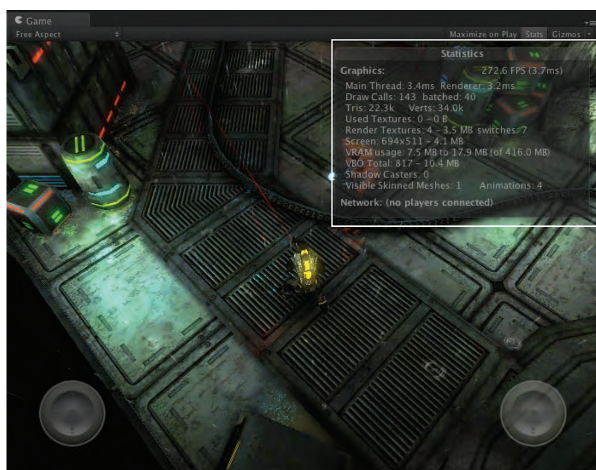


图2-29 显示状态

通过Gizmos工具栏，可勾选查看场景模型的所有工具信息。如图2-30所示，在Game视图右侧的勾选框中选择需要查看的工具信息。勾选相应工具后，表示Game视图中将显示其相关的信息，未勾选则表示Game视图中不显示它。



图2-30 Gizmos工具栏

使用Game视图控制条设置显示属性后，依然不会影响到游戏最终发布的效果。它只是用于当前编辑器Game视图窗口中的显示，此外还主要在开发过程中使用。

2.6.3　导出与导入

工程的导出好比将整个游戏工程打包，这样可以方便管理游戏工程。Unity可将游戏工程整体导出并且封装成“.unitypackage”包。导出工程包的方法如下：首先启动Unity，打开需要导出的游戏工程，然后在Unity导航菜单栏中选择“Assets”→“Export Package”菜单项，此时将弹出“Exporting package”界面，如图2-31所示。默认情况下，系统将勾选当前工程的所有项目资源。此时，也可在文件名左侧使用复选框进行动态删减操作。确保无误后，点击“Export”按钮，然后选择工程导出后保存的路径即可。

导入游戏工程包的方法非常简单：直接双击任意一个“.unitypackage”游戏工程包，系统将自动唤醒Unity并且自动打开选择的工程包。工程包导入后，选择打开对应的游戏场景文件，可以直接运行这个游戏项目。

导出与导入还有一个方便之处，那就是可在不同操作系统下完美实现导出与导入操作，比如在Mac操作系统下进行导出操作，然后将导出的.unitypackage包放在Windows操作系统下可直接进入导入。

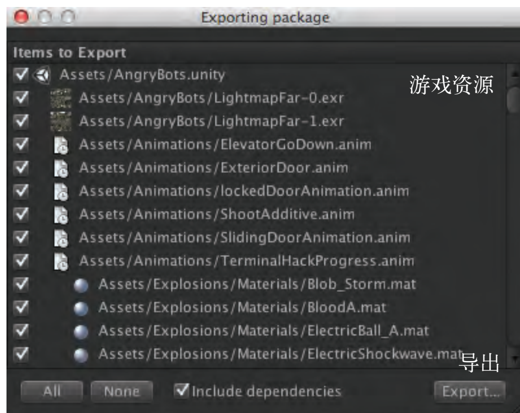


图2-31　“Exporting package”界面

2.7　第一个游戏实例（拓展训练）

学习到这里，我相信大家对Unity这套3D游戏引擎应该已经不是非常陌生了。为了让读者进一步了解Unity，我们将创建一些游戏对象与脚本来构建一个简单的3D游戏。在该示例中，我们使用GUI按钮来控制模型的移动与旋转。

为了让读者大概了解整个Unity 3D游戏开发过程，我们一步一步地来制作这个实例，具体操作步骤如下所示。

1. 创建第一个Unity 3D项目

打开Unity游戏编辑器，在导航菜单栏中选择“File”→“New Project”菜单项，然后在弹出

的“Project Wizard”窗口中选择“Create new Project”页面，我们将此项目命名为“CodeList_2_1”，确认无误后在窗口右下角处点击“Create Project”按钮完成这个项目的创建。

然后直接使用快捷键Command＋S保存场景文件（在Windows系统下，则是Windows徽标键＋S键）。默认情况下，项目创建完毕后，Unity会自动帮我们创建一个场景，并且将摄像机添加至其中。

2. 构建3D世界中的基本模型

在Hierarchy视图中分别创建游戏对象平面（Plane）、立方体（Cube）、球体（Sphere）、圆柱体（Cylinder）和胶囊体（Capsule），然后使用变换工具栏将它们摆放在合适的位置。

接着在游戏世界中添加一个光源属性。光源是非常重要的一个属性，一定要在游戏场景中设置它，如果不设置光源，Game视图会非常暗，严重影响游戏发布的效果。创建光源的方式如下：在Hierarchy视图中选择“Create”→“Directional Light”菜单项。如图2-32所示，该光源已经出现在Scene视图当中。下面需要旋转光源的角度使其投射在游戏场景中，好让摄像机更清晰地照射出3D世界中的模型。创建完所有模型后，使用移动工具将它们有规律地平放在平面之上。最后修改摄像机投射的位置，保证游戏对象全部映射在Game视图当中。

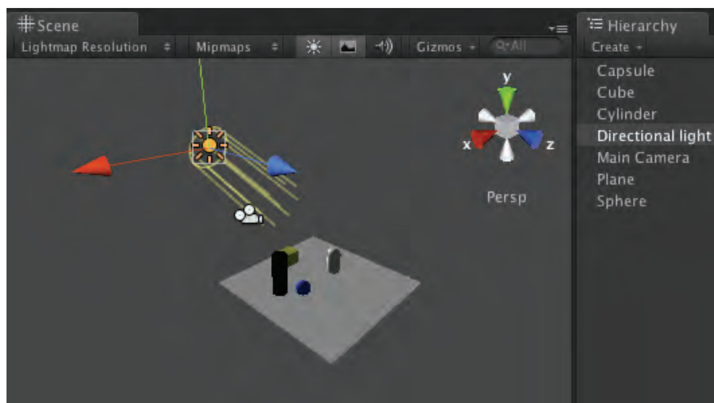


图2-32　创建模型

3. 使用脚本控制模型的移动

Unity共支持3种脚本语言——JavaScript、C#和BooScript，实际开发中普遍会使用C#或者JavaScript，本书也将以这两种语言来讲解。由于JavaScript相对比较简单并且更容易上手，所以推荐初学者使用JavaScript编写游戏脚本。本例先使用JavaScript脚本带领读者开始Unity开发。

在Project视图中点击“Create”→“JavaScript”菜单项来创建一条游戏脚本，暂时将其命名为“Script_02_01.js”，在该脚本中编写一段代码，具体如代码清单2-1所示。

代码清单2-1　Script_02_01.js文件

```
//模型移动速度
var TranslateSpeed = 20;
//模型旋转速度
var RotateSpeed = 1000;
```

```
//绘制UI
function OnGUI()
{
    //设置GUI背景颜色
    GUI.backgroundColor = Color.red;
    if(GUI.Button(Rect(10,10,70,30), "向左旋转"))
    {
        //向左旋转模型
        transform.Rotate(Vector3.up *Time.deltaTime * (-RotateSpeed));
    }

    if(GUI.Button(Rect(90,10,70,30), "向前移动"))
    {
        //向前移动模型
        transform.Translate(Vector3.forward * Time.deltaTime *TranslateSpeed);

    }

    if(GUI.Button(Rect(170,10,70,30), "向右旋转"))
    {
        //向右旋转模型
        transform.Rotate(Vector3.up *Time.deltaTime * RotateSpeed);
    }

    if(GUI.Button(Rect(90,50,70,30), "向后移动"))
    {
        //向后移动模型
        transform.Translate(Vector3.forward * Time.deltaTime * (-TranslateSpeed));
    }

    if(GUI.Button(Rect(10,50,70,30), "向左移动"))
    {
        //向左移动模型
        transform.Translate(Vector3.right * Time.deltaTime *(-TranslateSpeed));
    }

    if(GUI.Button(Rect(170,50,70,30), "向右移动"))
    {
        //向右移动模型
        transform.Translate(Vector3.right * Time.deltaTime * TranslateSpeed);
    }

    //显示模型位置信息
    GUI.Label(Rect(250, 10,200,30),"模型的位置" +transform.position);
    //显示模型旋转信息
    GUI.Label(Rect(250, 50,200,30),"模型的旋转" +transform.rotation);
}
```

本段代码中涉及的一些重要方法和属性如下所示。

- OnGUI()：此方法用于绘制GUI界面组件。
- GUI.Button()：设置一个按钮，返回true时表示该按钮被按下。
- GUI.Label()：设置一个文本框。

- ❑ `transform`：为当前绑定模型的变换对象。
- ❑ `transform.Rotate()`：设置模型旋转。
- ❑ `transform.Translate()`：设置模型平移。
- ❑ `Time.deltaTime`：该数值为一个只读属性，不可修改，表示完成最后一帧的时间，单位为秒。
- ❑ `Vector3`：标志一个模型移动或者旋转的方向。
- ❑ `Rect`：规定一个矩形区域，用于显示控件。

无论是 JavaScript脚本、C#脚本还是BooScript脚本，都可以使用拖曳的形式绑定到Hierarchy视图中任何一个游戏对象上，并且每一个游戏对象都可以绑定多条游戏脚本。

如图2-33所示，由于脚本已经编写完毕，下面我们需要将这条脚本绑定在对应的游戏对象上。在Project视图中选择脚本“Script_02_01”并将其拖曳到Hierarchy视图中的立方体（Cube）对象上，如果没有错误提示，表示这条脚本绑定成功，运行游戏后该游戏对象将执行该脚本中的内容。

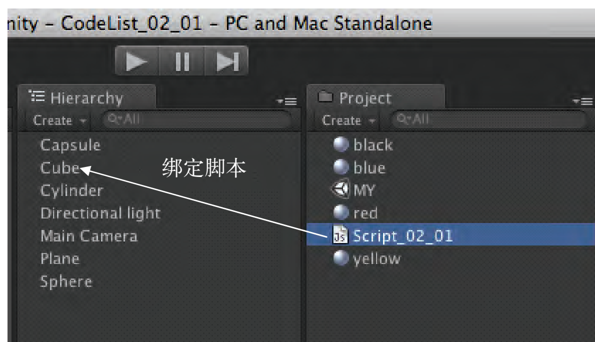


图2-33　绑定脚本

目前立方体对象与其他模型对象之间是不存在碰撞的，但是运行游戏后，可以控制立方体直接穿越另一个模型。为了让模型之间具有物理的碰撞，需要给该模型添加一个刚体（Rigidbody）属性。添加方法很简单，首先在Hierarchy视图中选中立方体对象，在Unity导航菜单栏中选择“Component”→“Physics”→“Rigidbody”菜单项即可（如图2-34所示）。



图2-34　添加刚体

实际上刚体与Unity物理引擎是紧密结合的，知识点也比较多、比较杂，读者一时半会也很难彻底理解。至于刚体具体起到什么作用，读者可以暂时理解为给某个模型添加了游戏物理引擎，使其可以感应物理的碰撞效果，第6章将详细介绍相关内容。

运行游戏后，可以在Game视图中通过选择方向按钮与旋转按钮来移动图2-35所示的立方体，同时会在游戏屏幕右上角将该模型的位置系数与旋转系数以一个文本框的形式显示出来。由于给立方体添加了刚体组件，所以移动立方体时，它将与其他模型发生碰撞。

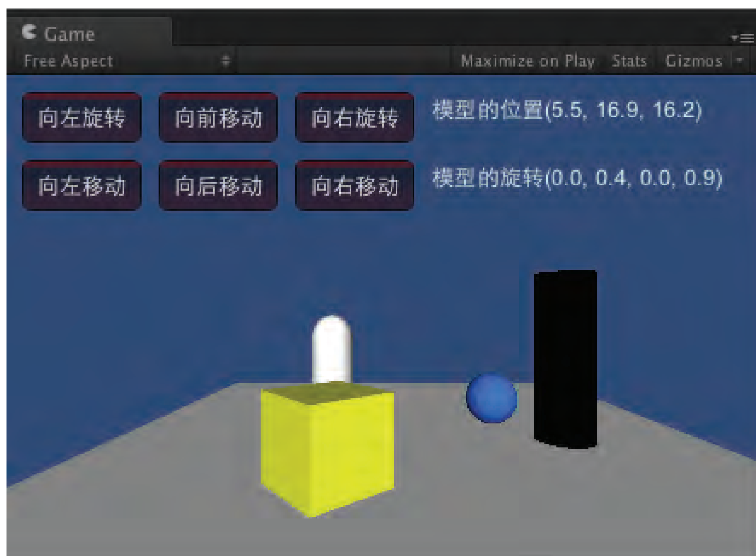


图2-35　游戏效果

2.8　本章小结

本章主要介绍了Unity编辑器的结构与使用方法，着重讲解了编辑器中的5大视图——Project视图、Hierarchy视图、Inspector视图、Scene视图和Game视图及其包含的所有控件，详细学习了这5大视图之间相互的工作原理以及它们之间的关系。在本章最后，以一个简单的游戏实例向读者详细介绍了使用Unity构建游戏的全过程。

Unity编辑器的结构是深入Unity开发的前提，希望读者可以认真学习这一章的内容，千里之行始于足下。在后面的章节中，我会深入浅出地带领大家系统学习Unity 3D游戏开发。

第3章

GUI游戏界面

在游戏的整个开发过程中，游戏界面设计占据非常重要的地位。因为游戏启动后，第一个映入玩家眼帘的就是游戏的UI界面。UI界面主要包括贴图、按钮和高级控件等。通常游戏界面的展现方式有很多种，大多数都由自定义图形界面组成。

Unity为开发者提供了一套非常完善的图形化界面引擎，它包括常见的游戏窗口、文本框、输入框、拖动条、按钮、贴图框等，无论是做软件还是做游戏，都可以很方便地使用。

另外，Unity提供了界面自定义皮肤的功能。控件不仅可以使用默认的皮肤，还可以自定义皮肤。自定义皮肤不仅可以美化游戏界面，还可以提升游戏品质。Unity游戏界面主要由GUI完成。在本章中，我们将使用JavaScript脚本向读者详细介绍Unity中有关GUI界面的所有高级控件。

3.1 GUI 高级控件

系统高级UI控件已经成为游戏开发中不可缺少的一部分。高级界面由系统提供，所以运行效率要远远高于低级界面（高级界面为系统实现，低级界面为自己手动实现）。拿按钮控件来说吧，不使用系统提供的按钮控件，我们也可以使用低级界面模拟实现按钮的功能。不过低级界面实现的“按钮”没有系统高级界面实现的按钮效率高，但是低级界面制作的“按钮”比较灵活，可以任意修改。

GUI高级控件的种类非常繁多，包括标签、按钮、输入框和拖动条等。它们可用于任何游戏或软件的界面研发。GUI高级控件的应用也非常广泛，比如网络游戏中输入账号与密码的提示框，通关游戏后上传游戏积分的按钮，创建角色时输入的角色信息等。下面将分别向读者介绍GUI高级控件的相关用法。

3.1.1 Label控件

使用Label控件（标签控件），可以在游戏界面中以文本的形式展示出一段字符串信息。使用Label控件，我们不仅可以输入字符串，还可以贴图。

由于Unity的脚本默认编码不支持中文输出（中文会以“？”的形式显示），所以显示中文时需要修改脚本的编码格式，如在JavaScript脚本中将编码格式修改为“UTF-8”即可。本例我们将字符串“`HelloWorld!`”、屏幕宽高和贴图显示在屏幕中，具体代码见代码清单3-1。

代码清单3-1　Script_03_01.js文件

```
//接收外部赋值字符串
var str : String;
//接收外部赋值贴图
var imageTexture : Texture;
//贴图宽度
private var imageWidth : int;
//贴图高度
private var imageHeight : int;
//当前屏幕高度
private var screenWidth : int;
//当前屏幕宽度
private var screenHeight : int;

function Start()
{
    //得到屏幕宽高
    screenWidth = Screen.width;
    screenHeight = Screen.height;
    //得到图片宽高
    imageWidth = imageTexture.width;
    imageHeight = imageTexture.height;
}

function OnGUI()
{
    //将文字内容显示在屏幕中
    GUI.Label(Rect(100, 10, 100, 30), str);
    GUI.Label(Rect(100, 40, 100, 30), "当前屏幕宽:" + screenWidth);
    GUI.Label(Rect(100, 80, 100, 30), "当前屏幕高:" + screenHeight);
    //将贴图显示在屏幕中
    GUI.Label(Rect(100, 120, imageWidth, imageHeight),imageTexture);
}
```

本例中，我们首先学习两个系统提供的方法。第一个是`Start()`方法，该方法只执行一次，所以需要将初始化的相关代码都放在`Start()`方法中。第二个是`OnGUI()`方法，它是界面绘制方法，所有GUI的绘制都需要在这个方法中实现。

脚本只有绑定在对象身上时，才会执行它自身的生命周期。由于新创建的游戏场景默认会添加摄像机对象，那么我们将这条脚本绑定在摄像机对象中。绑定脚本的方法很简单，只需在Project视图中用鼠标选择脚本文件，然后直接将其拖入Hierarchy视图中的摄像机对象中即可。绑定完毕后运行游戏，将自动执行脚本生命周期，此时即可看到界面效果。

此外，可在Inspector视图中为脚本中的公用变量赋值，如图3-1所示。选择摄像机后，将Project视图中的贴图对象0拖动至右侧的“Image Texture”变量中，在“Str”变量中写入“HelloWorld!”，执行脚本便可将在这里赋值的变量传递至程序中。

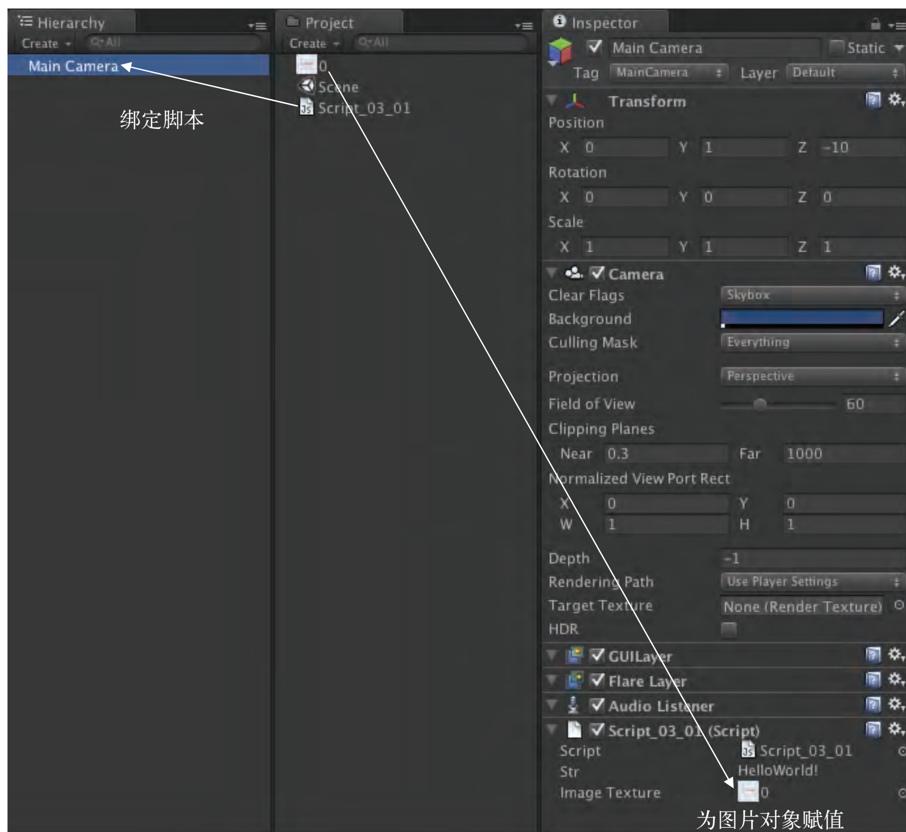


图3-1 连接变量

注意 只有公有变量才可以在编辑器中以拖曳对象或输入的形式赋值。在声明变量时，在变量前方添加public关键字或未添加任何关键字表示该变量为公有变量，如：

```
public var str : String;
var str : String;
```

在变量前方添加private关键字的变量为私有变量。私有变量不会出现在编辑器中，只能在脚本中使用，如：

```
private var str : String;
```

运行程序后的效果如图3-2所示，可以发现，文本与贴图均显示在游戏视图当中。

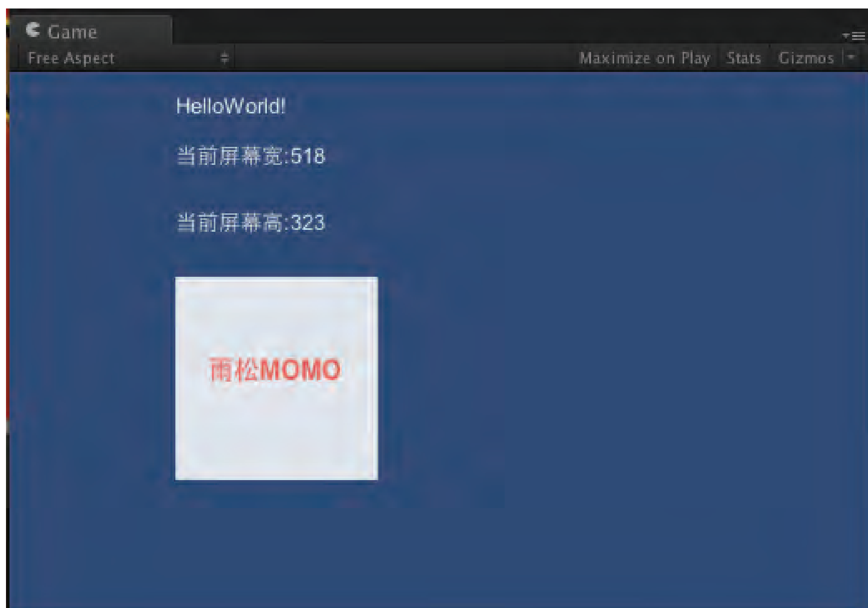


图3-2　Label控件演示效果图

3.1.2　Button控件

在开发中，Button控件（按钮控件）是十分常见的控件之一，可以用来判断用户在程序中的一些操作行为，比如对话框中的“确定”和“取消”按钮。

按钮共有3个基本状态组成：未点击状态、击中状态、点击后状态。一般情况下，游戏界面的按钮只监听“未点击状态”与“点击后状态”。

按照展现方式，按钮可以分为两种：“普通按钮”和“图片按钮”。普通按钮为系统默认显示的按钮，而图片按钮可以设定按钮的背景图案。

图3-3设置了三个按钮，其中一个为图片按钮，一个为文字按钮，最后一个是连续按钮。此外，还设置了按钮的文字颜色与背景颜色，并且监听按钮点击的事件并且在游戏屏幕中以文本框的形式显示用户点击的是图片按钮还是文字按钮。代码中还添加了一个时间计数器，当用户点击连续按钮后，计数器会记录该按钮按下时的时间，示例代码如代码清单3-2所示。

代码清单3-2　Script_03_02.js文件

```
//按钮贴图
var buttonTexture : Texture2D;

//提示信息
private var str : String;
```

```
//时间计数器
private var frameTime : int;

function Start()
{
    //初始化赋值
    str = "请您点击按钮";
}

function OnGUI()
{
    //显示提示信息内容
    GUI.Label(Rect(10, 10, Screen.width, 30), str);


    if(GUI.Button(Rect(10,50,buttonTexture.width,buttonTexture.height),buttonTexture)){
        //点击按钮修改提示信息
        str = "您点击了图片按钮";
    }
    //设置按钮中文字的颜色
    GUI.color = Color.green;
    //设置按钮的背景色
    GUI.backgroundColor = Color.red;

    if (GUI.Button(Rect(10,200,70,30),"文字按钮")){
        //点击按钮修改提示信息
        str = "您点击了文字按钮";
    }

    //设置按钮中文字的颜色
    GUI.color = Color.yellow;
    //设置按钮的背景色
    GUI.backgroundColor = Color.black;

    if (GUI.RepeatButton(Rect(10,250,100,30),"按钮按下中")){

        //点击按钮修改提示信息
        str = "按钮按下中的时间："+ frameTime;
        //时间计数器
        frameTime++;
    }

}
```

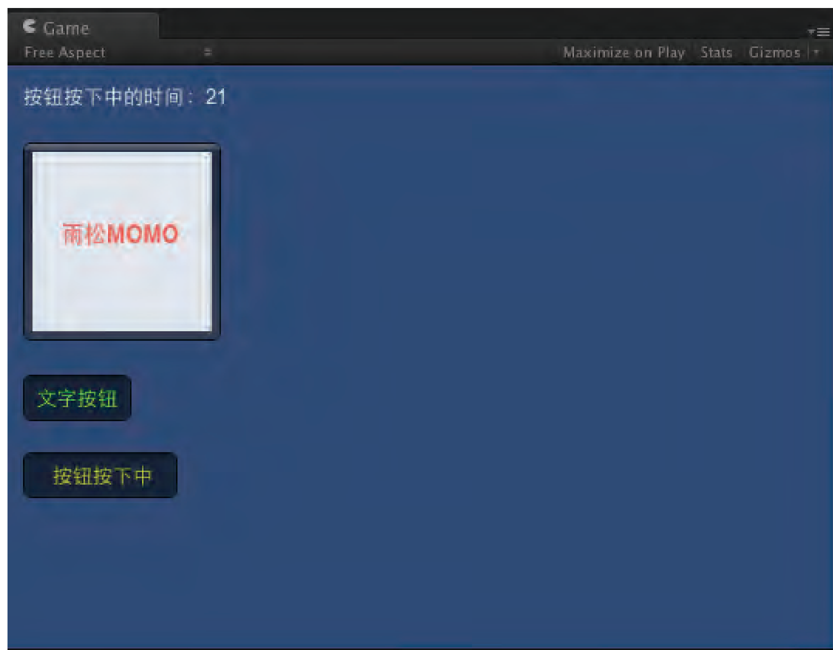


图3-3　Button控件演示效果图

在该示例中，我们使用GUI.Button()方法添加按钮，当该方法返回true时，表示按钮被按下。而使用GUI.RepeatButton()方法可添加一个连续按钮，用于监听该按钮按下中的状态，当该方法返回true时，表示按钮处于按下中。另外，使用GUI.color可设置文字的颜色，使用GUI.backgroundColor可设置按钮的背景色。

3.1.3　TextField控件

TextField控件主要用于监听用户输入的信息，其应用非常普遍，比如在游戏登录界面中，玩家输入用户名与密码后，点击“确认”按钮判断其输入是否正确，或者游戏通关后填写胜利者姓名与输入相关游戏信息等。

一般情况下，使用GUI.TextField()方法显示输入框，该方法的返回值为用户输入的字符串信息。使用GUI.PasswordField()方法，可以将用户输入的信息显示为任意字符串，一般在输入密码时将密码以“****”的形式显示。后面的参数"*"[0]用来将输入的字符串显示为“*****”。

下面举例说明该控件的用法，效果如图3-4所示。在本例中，当用户输入用户名与密码后，点击“登录”按钮，程序将获取输入框中用户输入的信息，并且将这些信息显示在屏幕最上方，具体代码如代码清单3-3所示。

提示 获取输入信息时，只能获取到输入的字符串，因此在必要的时候需要进行强制转换。比如，若账号由纯数字组成，但用户却输入了非数字，比如字母或汉字，那么将字符串强制转换为整型时，肯定会抛出异常，所以在强制转换输入框中获取的字符串时，需要捕获异常，避免出错。

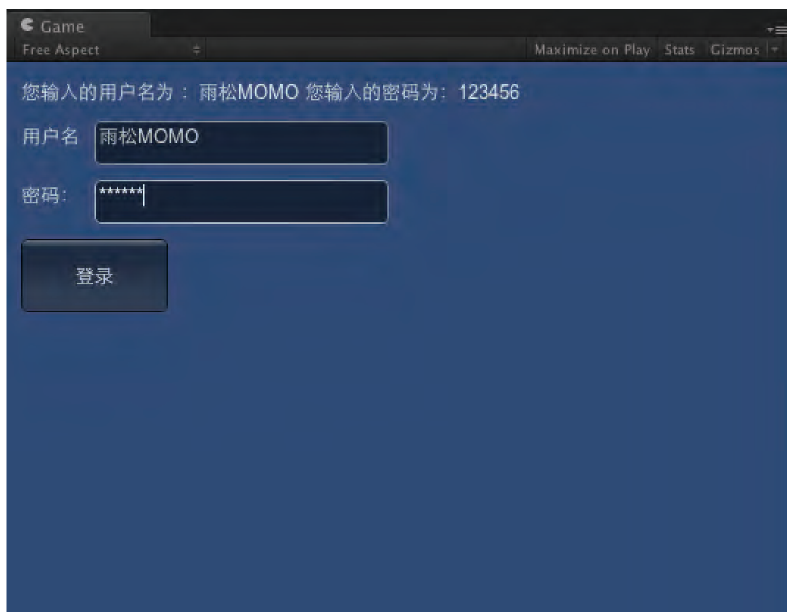


图3-4 TextField控件演示效果图

代码清单3-3 Script_03_03.js文件

```
//用户名
private var editUsername : String;
//密码
private var editPassword : String;
//提示信息
private var editShow : String;

function Start()
{
    editShow = "请您输入正确的用户名与密码";
    editUsername = "请输入用户名";
    editPassword = "请输入密码";
}

function OnGUI()
{
```

```
        //显示提示信息内容
        GUI.Label(Rect(10, 10, Screen.width, 30), editShow);

        if (GUI.Button(Rect(10,120,100,50),"登录"))
        {
            //点击按钮修改提示信息
            editShow = "您输入的用户名为 : " + editUsername + " 您输入的密码为: "+ editPassword;
        }
        //编辑框提示信息
        GUI.Label(Rect(10, 40, 50, 30), "用户名");

        GUI.Label(Rect(10, 80, 50, 30), "密码: ");

        //获取输入框输入的内容
        editUsername = GUI.TextField (Rect (60, 40, 200, 30), editUsername, 15);
        editPassword = GUI.PasswordField (Rect (60, 80, 200, 30), editPassword,"*"[0],15);
    }
```

在上述代码中，我们使用`GUI.TextField()`方法在屏幕中添加一个输入框，用户可在输入框中输入任意文字。该方法的返回值为用户输入的信息，返回类型为`String`。`GUI.PasswordField()`方法同样可以在屏幕中添加一个输入框，其用法和`GUI.TextField()`几乎一样，但是这个输入框限制了显示的内容，比如提示输入密码时，输入的所有内容将以“*”的形式显示在输入框当中。

3.1.4　ToolBar控件

ToolBar控件用于创建工具栏，并且以Tab页面的形式来展现，选择其中任意一项后即可返回所选项的ID。

工具栏常位于界面顶部或者底部，其中每个按钮可以使用贴图的形式展现。如图3-5所示，我们在界面顶部实现了一个工具栏，其中放置着4个按钮，通过点击不同按钮来回切换界面显示的内容，并且内容以单选项的形式呈现出来。本例中涉及的方法有两个，其中`GUI.Toolbar()`方法用于创建一个工具栏，`GUI.Toggle()`方法用于创建一个单选控件，具体代码如代码清单3-4所示。

代码清单3-4　Script_03_04.js文件

```
//工具栏选择按钮的ID
private var select : int;

//工具栏显示按钮的字符串
private var barResource : String[];

//选择按钮是否被按下
private var selectToggle0 : boolean;
private var selectToggle1 : boolean;

function Start()
{
```

```
        //初始化
        select = 0;
        barResource = ["第一个工具栏","第二个工具栏","第三个工具栏","第四个工具栏"];

        selectToggle0 = false;
        selectToggle1 = false;
    }

    function OnGUI()
    {
        //备份上一次工具栏选择的ID
        var oldSelect = select;
        //重新计算本次工具栏选择的ID
        select = GUI.Toolbar(Rect (10, 10, barResource.length * 100, 30), select,
            barResource);
        //如果两次选择的是不同的工具栏，将选择按钮全部释放掉
        if(oldSelect != select)
        {
            selectToggle0 = false;
            selectToggle1 = false;
        }

        //根据工具栏选择的ID显示不同的信息
        switch(select)
        {
        case 0:
            selectToggle0 = GUI.Toggle(Rect(10, 50, 150, 30), selectToggle0, "第一个工具
                栏单项选择——1");
            selectToggle1 = GUI.Toggle(Rect(10, 80, 150, 30), selectToggle1, "第一个工具
                栏单项选择——2");
            break;
        case 1:
            selectToggle0 = GUI.Toggle(Rect(10, 50, 150, 30), selectToggle0, "第二个工具
                栏单项选择——1");
            selectToggle1 = GUI.Toggle(Rect(10, 80, 150, 30), selectToggle1, "第二个工具
                栏单项选择——2");
            break;
        case 2:
            selectToggle0 = GUI.Toggle(Rect(10, 50, 150, 30), selectToggle0, "第三个工具
                栏单项选择——1");
            selectToggle1 = GUI.Toggle(Rect(10, 80, 150, 30), selectToggle1, "第三个工具
                栏单项选择——2");
            break;
        case 3:
            selectToggle0 = GUI.Toggle(Rect(10, 50, 150, 30), selectToggle0, "第四个工具
                栏单项选择——1");
            selectToggle1 = GUI.Toggle(Rect(10, 80, 150, 30), selectToggle1, "第四个工具
                栏单项选择——2");
            break;
        }


    }
```

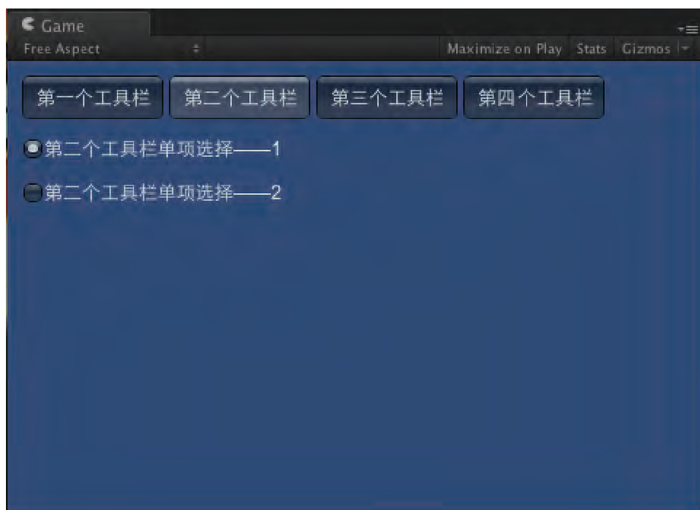


图3-5　ToolBar控件演示示例效果图

在上述代码中，`GUI.Toolbar()`方法的返回值表示在ToolBar控件中选择的ID，而`GUI.Toggle()`方法的返回值表示单选后的ID。

3.1.5　Slider控件

Slider控件由滑块与滑动条组成。使用Slider控件，可以计算出滑块在滑动过程中占整个滑动条的比例。如果滑动条的整体长度为100，则滑块滑动的范围就是0至100。

按照展示方式，滑动条可分为两种：一种为水平滑动条（`HorizontalSlider`），另一种为垂直滑动条（`VerticalSlider`），它们之间的用法完全相同。

在开发中，我们常使用滑动条来调节音量或者颜色等。下面举例说明Slider控件的用法，演示代码如代码清单3-5所示。

代码清单3-5　Script_03_05.js文件

```
//纵向滑动条数值
var verticalValue : int = 0;

//横向滑动条数值
var horizontalValue : float = 0.0f;

function OnGUI()
{
    //计算滑动进度
    verticalValue = GUI.VerticalSlider (Rect (25, 25, 30, 100), verticalValue, 100, 0);
    horizontalValue = GUI.HorizontalSlider(Rect(50, 25, 100, 30), horizontalValue,
        0.0f, 100.0f);
    //将滑动进度显示在屏幕中
```

```
    GUI.Label(Rect(10, 150, Screen.width, 30), "纵向滑动条当前进度：" + verticalValue +"%");
    GUI.Label(Rect(10, 180, Screen.width, 30), "横向滑动条当前进度： " + horizontalValue
        +"%");
}
```

运行上述代码，效果如图3-6所示。

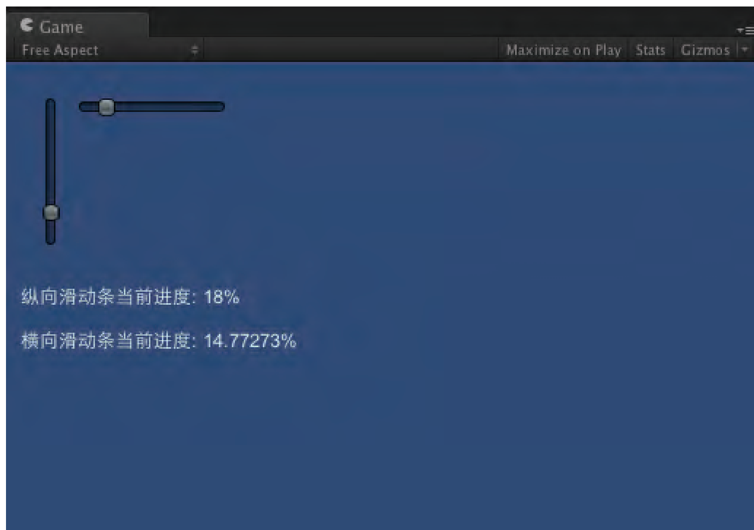


图3-6　Slider控件演示效果图

在该示例中，主要使用了如下两个重要方法。

- `GUI.VerticalSlider()`：用于纵向滑块，它的第一个参数表示滑动条显示的范围，第二个参数表示当前滑动的数值，第三个参数表示滑动条起点的数值，第四个参数表示滑动条终点的数值。
- `GUI.HorizontalSlider()`：用于横向滑块，具体参数和`GUI.VerticalSlider()`类似。

3.1.6　ScrollView控件

如果游戏界面中的GUI控件过多，超出了屏幕的显示范围，就需要使用ScrollView控件来完成它的展示效果。

ScrollView控件可设定一个滚动显示区域。如果横向或纵向的GUI控件超出了其显示区域，视图下方或者右方将会出现滚动条。在开发中使用ScrollView控件的情况非常普遍，比如如果游戏中帮助信息或者关于信息过长，就可以使用滚动条来查看相关信息。

下面举例说明ScrollView控件的用法，效果如图3-7所示。本例实现了一个较大的游戏视图，分别拖动下面与右面的滚动条来整体拖动该视图，具体代码如代码清单3-6所示。



图3-7　ScrollView控件演示示例

代码清单3-6　Script_03_06.js文件

```
//滚动条位置
var scrollPosition : Vector2;

function Start()
{
    //初始化滚动条位置
    scrollPosition[0] = 50;
    scrollPosition[1] = 50;

}

function OnGUI() {
    //开始滚动视图
    scrollPosition = GUI.BeginScrollView (Rect (0,0,200,200),scrollPosition, Rect (0,
        0, Screen.width, 300),true,true);

    GUI.Label(Rect(100, 40, Screen.width, 30), "测试滚动视图，测试滚动视图，测试滚动视图，
        测试滚动视图。");

    //结束滚动视图
    GUI.EndScrollView();

}
```

在上述示例代码中，`Start()`方法用于设置默认情况下滚动条的位置，数组`scroll-Position[0]`表示滚动视图横向滚动滑块位置，数组`scrollPosition[1]`表示滚动视图纵向滚动滑块位置。

将这两个数组全部存储在`Vector2`中，拖动滚动条后，在程序内存中会动态修改两个滚动条的位置。`GUI.BeginScrollView()`方法用于开始滚动视图，该方法的第一个参数用于设置滚动

显示视图的范围，第二个参数用于设置视图滚动条的起始位置，第三个参数用于设置滚动整体显示范围（这里需要说明一下滚动视图的显示范围必须小于游戏视图整体范围），第四个参数与第五个参数为true时，表示内容超过滚动显示范围后显示滚动条，否则不显示滚动条。

另外，需要注意的是，GUI.BeginScrollView()与GUI.EndScrollView()必须成对出现（前者表示滚动视图开始，后者表示滚动视图结束），否则会抛出异常。

3.1.7 群组视图

群组视图（GroupView控件）可将多个视图全部放在一个群组当中。将视图添加进群组当中后，群组中任何视图的坐标都是相对坐标，它是相对群组视图左上角的坐标。

修改群组视图的坐标后，群组中所有视图的坐标都会跟着修改。推荐使用群组视图来制作游戏界面，因为设备的屏幕尺寸不同，这样做可以避免对坐标进行多次修改的麻烦。在群组视图中，使用GUIContent()方法可设置提示信息，使用GUI.tooltip可以得到GUIContent()方法中第二个字符串参数设置的提示字符串。

下面举例说明群组视图的用法，效果如图3-8所示。本例共添加了两个群组视图，并且为每一个视图添加了图片、按钮和标签组件，代码如代码清单3-7所示。



图3-8 GroupView控件演示示例

代码清单3-7 Script_03_07.js文件

```
//贴图
var viewTexture0 : Texture2D;
var viewTexture1 : Texture2D;
```

```
function OnGUI()
{

    //开始这个群组
    GUI.BeginGroup(new Rect(10, 50, 200, 400));
    //显示贴图，坐标为相对群组的点(10, 50)的坐标
    GUI.DrawTexture(Rect(10,20,viewTexture0.width,viewTexture0.height),
        viewTexture0);
    //标签提示信息
    GUI.Label(Rect(10,150,100,40),"群组视图1");
    //按钮
    GUI.Button(Rect(10,190,100,40),"按钮");
    //结束这个群组
    GUI.EndGroup();

    //开始这个群组
    GUI.BeginGroup(new Rect(300, 0, 500, 400));
    //显示贴图，坐标为相对群组的点(300, 0)的坐标
    GUI.DrawTexture(Rect(10,20,viewTexture1.width,viewTexture1.height),
        viewTexture1);
    //标签提示信息
    GUI.Label(Rect(10,150,100,40), "群组视图2");
    //按钮
    GUI.Button(Rect(10,190,100,40),"按钮");
    //结束这个群组
    GUI.EndGroup();

}
```

使用`GUI.BeginGroup()`方法可以创建一个群组视图，但是必须以`GUI.EndGroup()`方法结束群组视图。在`GUI.BeginGroup()`方法中可设定群组视图的区域，在该区域中可添加任意控件，如果超出该范围，则不予显示。另外，群组视图中所有控件的坐标都采取相对坐标，相对该群组视图左上角的坐标。它的好处在于移动群组视图后，其中的所有控件都会跟着移动，永远保持相对的位置。

本例一共创建了两个群组视图，使用`GUI.DrawTexture()`方法来绘制视图中的贴图，该方法的第一个参数表示贴图的绘制区域，第二个参数表示贴图的资源。

3.1.8　窗口

窗口在游戏开发中并不陌生，所有视图都需要依赖窗口来显示，我们可以把窗口理解为视图的父类。前面我们介绍了各式各样的游戏视图，它们都属于窗口的子类。游戏界面可以由若干个窗口组成，窗口又由若干个视图组成。

创建窗口时需要设定它的显示区域，在窗口中可以添加任意组件，前提是组件的显示区域必须在窗口当中，否则无法显示。另外，窗口中所有控件的坐标均采取相对坐标，相对窗口左上角的坐标。下面举例说明窗口的用法，效果如图3-9所示。本例在屏幕中分别绘制两个窗口，在这两个窗口中均添加了Box组件与Button组件，并且监听按钮点击事件，具体代码如代码清单3-8所示。

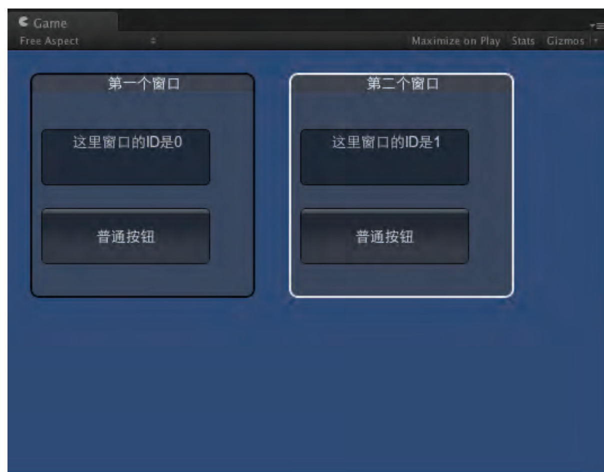


图3-9 窗口

代码清单3-8 Script_03_08.js文件

```
//默认窗口位置
private var window0 : Rect = Rect (20, 20, 200, 200);
private var window1 : Rect = Rect (250, 20, 200, 200);
function OnGUI()
{
    //在这里注册两个窗口
    GUI.Window (0, window0, oneWindow, "第一个窗口");
    GUI.Window (1, window1, twoWindow, "第二个窗口" );
}
//显示窗口1的内容
function oneWindow (windowID : int)
{

    GUI.Box(Rect(10,50,150,50),"这里窗口的ID是" + windowID);
    if(GUI.Button(Rect(10,120,150,50),"普通按钮"))
    {
        Debug.Log("窗口ID = "+windowID+"按钮被点击");
    }

}

//显示窗口2的内容
function twoWindow (windowID : int)
{
    GUI.Box(Rect(10,50,150,50),"这里窗口的ID是" + windowID);
    if(GUI.Button(Rect(10,120,150,50),"普通按钮"))
    {
        Debug.Log("窗口ID = "+windowID+"按钮被点击");
    }
}
```

在上述代码中，我们使用`GUI.Window()`方法注册窗口，该方法共有4个参数，第一个参数表示窗口的ID，第二个参数表示窗口显示的区域，第三个参数表示一个回调方法名称，窗口中包含的所有视图控件将写入这个方法，第四个参数表示窗口的标题名称。

3.1.9　GUI Skin

通过之前章节的学习，我想大家已经掌握了Unity大部分的GUI控件，但是直接使用这些控件开发游戏还远远不够，因为系统默认的界面实在过于粗糙与单调。为了让自己的游戏界面活灵活现，我们需要使用GUI Skin为控件添加一个漂亮的皮肤。

首先在Project视图中点击“Create”→“GUI Skin”菜单项，创建一个GUI Skin。使用GUI Skin，可以修改任何系统提供的控件皮肤。如图3-10所示，在Inspector视图中可以清晰地看到GUI Skin可设置的皮肤控件，下面对其进行简要介绍。

- Font：可设置系统字体或者自定义字体。
- Box：可设置盒子的显示皮肤。
- Button：可设置按钮的显示皮肤。
- Toggle：可设置选择框的显示皮肤。
- Label：可设置文本框的显示皮肤。
- Text Field：可设置输入框的显示皮肤。
- Text Area：可设置多行输入框的显示皮肤。
- Window：可设置窗口的显示皮肤。
- Horizontal Slider：可设置水平滚动条的显示皮肤。
- Horizontal Slider Thumb：可设置水平滚动条上滑动块的显示皮肤。
- Vertical Slider：可设置垂直滑动条的显示皮肤。
- Vertical Slider Thumb：可设置垂直滑动条上滑动块的显示皮肤。
- Horizontal Scrollbar：可设置水平滚动条的显示皮肤。
- Horizontal Scrollbar Thumb：可设置水平滚动条上滑块的显示皮肤。
- Horizontal Scrollbar Left Button：可设置水平滚动条左边按钮的显示皮肤。
- Horizontal Scrollbar Right Button：可设置水平滚动条右边按钮的显示皮肤。
- Vertical Scrollbar：可设置垂直滚动条的显示皮肤。
- Vertical Scrollbar Thumb：可设置垂直滚动条上滑块的显示皮肤。
- Vertical Scrollbar Up Button：可设置垂直滚动条上边按钮的显示皮肤。
- Vertical Scrollbar Down Button：可设置垂直滚动条下边按钮的显示皮肤。
- Scroll View：可设置滚动视图的显示皮肤。
- Custom Styles：自定义风格显示皮肤。
- Settings：一些其他设置。

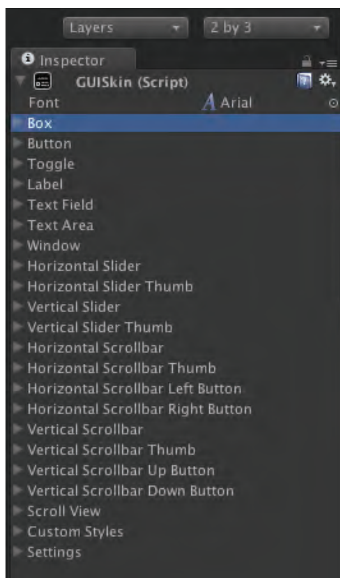


图3-10 界面皮肤

在列表中展开各个皮肤控件后，我们发现它们的设置选项基本上都是相同的，下面以Button为例进行介绍。展开Button控件后，按钮的所有皮肤属性出现在我们面前（如图3-11所示），修改这些属性便可设置自定义皮肤。

下面先简要介绍一下Button控件中各个皮肤属性的含义。

- Name：控件的名称。
- Normal：设置文字默认显示颜色和背景颜色。
- Hover：设置停留状态显示颜色和背景颜色，可用于鼠标停留在按钮、输入框、选择框等相关控件上但是没有点击时的显示。
- Active：设置激活状态显示颜色和背景颜色，用于按钮或者选择框点击后的显示。
- Focused：获得焦点状态，用于窗口得到焦点后的显示。
- On Normal：默认状态，未选中状态，用于选择框控件显示的内容。
- On Hover：停留状态，用于选择框控件选中后文字的显示。
- On Active：激活状态，用于选择框控件选中时文字的显示。
- On Focused：获得焦点状态。
- Border：处理边界，它不会影响在按钮平面显示的宽高。
- Padding：设置按钮显示的内容与按钮边缘的偏移位置。
- Margin：设置按钮整体的偏移位置。
- Overflow：设置按钮超出原有大小的距离。
- Font：设置针对该控件的字体。

- ❑ Image Position：设置图片位置。
- ❑ Alignment：设置内容对齐方式。
- ❑ Word Wrap：是否自动换行。
- ❑ Text Clipping：设置文本的剪切格式。
- ❑ Content Offset：设置内容的偏移量。
- ❑ Fixed Width：设置边缘固定的宽度。
- ❑ Fixed Height：设置边缘固定的高度。
- ❑ Font Size：字体的大小，默认大小为0。
- ❑ Font Style：字体的风格。
- ❑ Stretch Width：是否伸展宽度。
- ❑ Stretch Height：是否伸展高度。



图3-11　Button控件的皮肤属性

下面举例说明GUI Skin的用法，效果如图3-12所示。在本例中，我们设置了一些控件自定义皮肤的显示颜色以及应用控件的普通状态、停留状态和激活状态，详细代码如代码清单3-9所示。

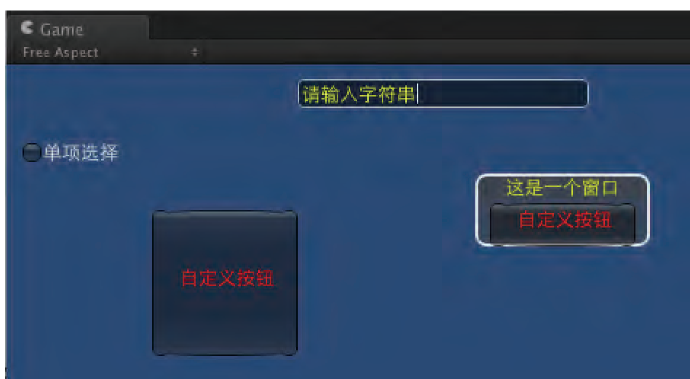


图3-12 GUI Skin效果图

代码清单3-9 Script_03_09.js文件

```
//自定义皮肤
var mySkin : GUISkin;
//单选控件是否被选中
private var choose : boolean = false;

//拖动窗口的位置
var windowRect : Rect = Rect (20, 20, 120, 50);

//输入框中默认显示的内容
var edit : String = "请输入字符串";

function OnGUI()
{
    //设置GUI皮肤为自定义皮肤
    GUI.skin = mySkin;
    //绘制自定义按钮
    GUI.Button(Rect (100,100,100,100),"自定义按钮");

    //单项选择
    choose = GUI.Toggle(Rect(10, 50, 100, 30), choose, "单项选择");

    //输入框
    edit = GUI.TextField (Rect (200, 10, 200, 20), edit, 25);

    //注册窗口
    windowRect = GUI.Window (0, windowRect, setWindow, "这是一个窗口");

}
//创建窗口内容
function setWindow (windowID : int)
{
    //创建一个可以自由拖动的窗口
    GUI.DragWindow();
    //绘制自定义按钮
    GUI.Button(Rect (10,20,100,30),"自定义按钮");
}
```

在上述代码中，我们使用`GUI.skin = mySkin`为当前GUI的皮肤赋值，而`mySkin`为外部创建的自定义皮肤。在Unity编辑器中，使用拖曳的方式将自定义皮肤赋予代码中的`mySkin`皮肤对象即可。

3.1.10　自定义风格组件

自定义风格组件可设置一组特殊的组件。虽然系统已经提供了很多高级控件，但还是无法满足项目中的一些特殊需求，此时可以通过自定义风格组件来实现。下面还是用按钮作为例子，首先使用系统提供的自定义皮肤修改按钮的样式，然后使用程序在GUI中开始绘制，无论绘制多少按钮，它们的样式都完全一样，因为Button组件是系统组件，所以修改它将适用于所有新添加的按钮。如果我们想将这些按钮的样式全部区分开，就需要使用Unity提供的自定义风格组件，因为它可以设定某一类特定的组件。

首先在Project视图中选择“Create”→“GUI Skin”菜单项，创建一个自定义皮肤资源，在右侧的Inspector视图中展开Custom Styles下拉菜单项（如图3-13所示），然后可直接在“Size”输入框中修改自定义风格组件的数量。本例中设置了2个自定义风格组件，分别取名为“Custom0”与“Custom1”，在代码中根据控件的名称就可以拿到不同的组件对象。

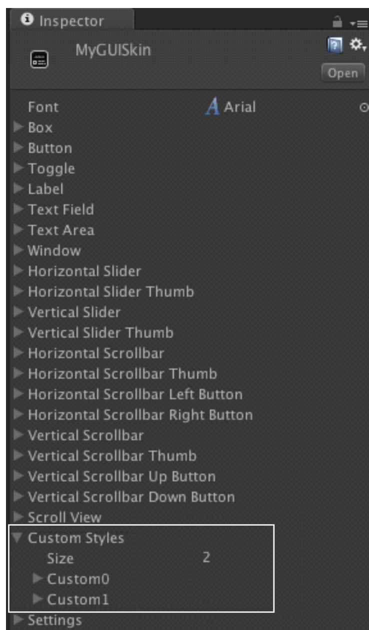


图3-13　自定义风格组件

下面我们给这两个自定义风格组件添加贴图。如图3-14所示，在“Custom0”与“Custom1”组件的普通状态、停留状态和激活状态下分别添加不同的贴图。

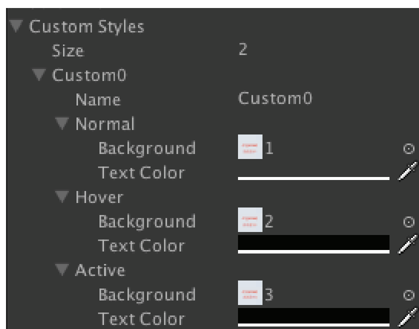


图3-14　设定组件资源

然后使用代码在GUI中将这两个组件一起绘制在屏幕当中，如图3-15所示。因为在组件的不同状态设置了不同的贴图，所以选中与未选中状态下组件的贴图将完全不一样。

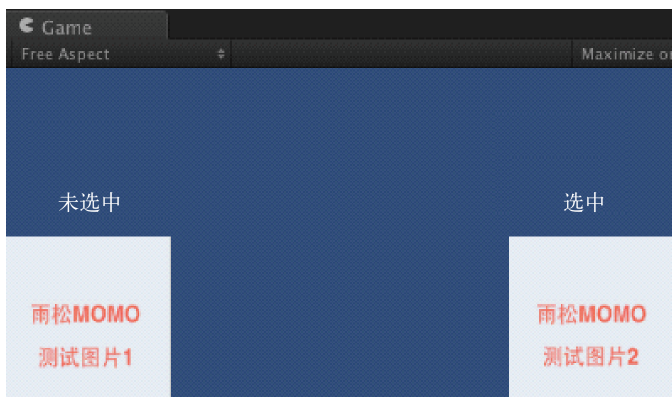


图3-15　自定义风格组件示例效果图

设置完自定义风格组件后，需要将自定义的皮肤资源赋给脚本中的GUISkin对象，具体代码如代码清单3-10所示。

代码清单3-10　Script_03_10.js文件

```
//自定义皮肤
var mySkin : GUISkin;

function OnGUI()
{
    //设置GUI皮肤为我们自定义的皮肤
    GUI.skin = mySkin;
    //绘制按钮，通过名称找到对应风格
    GUI.Button(Rect (0,100,300,100),"","Custom0");
    GUI.Button(Rect (300,100,300,100),"","Custom1");
}
```

在上述代码中，在绘制自定义组件时，我们调用了`GUI.Button()`方法，该方法共有3个参数，第一个参数表示按钮显示的位置，第二个参数表示按钮显示的文字，第三个参数表示自定义风格组件的名称，该名称应和图3-14中的名称对应。

3.2 GUILayout游戏界面布局

游戏界面的制作效果有很多种，有复杂绚丽的界面，也有简单明了的界面，而设计方式是仁者见仁，智者见智。在跨平台游戏界面开发中，最麻烦的事就是各个游戏平台的分辨率不一样，甚至相同平台的分辨率也不一样，这无疑给移植造成非常大的麻烦。因此，在制作游戏界面时，使用绝对坐标值是相当危险的一件事。因为如果跨平台移植的话，分辨率发生了改变，开发者就得为其重新计算坐标，这在开发效率上将大打折扣。为了避免后期对坐标重新进行计算，前期制作界面时可以考虑自适应屏幕布局。GUI为开发者提供了游戏布局的概念，并且在布局当中所有的坐标点都是相对坐标，所以使用GUI游戏界面布局来制作界面将更有效地实现自适应屏幕。

3.2.1 GUI与GUILayout的区别

通过之前的学习，我相信大家对GUI应该并不陌生了，那么GUILayout是什么东西呢？它是游戏界面的布局。从命名中就可以看到这两个东西非常相像，但是在使用过程中两者还是存在一定区别的。

使用GUI绘制控件的时候，需要设置控件的`Rect()`方法，也就是说需要设定控件的整体显示区域。这样设置的控件非常不灵活，因为它的坐标以及大小已经固定死了，这时如果控件中的内容长度发生改变，就会直接影响展示效果。例如，在界面中绘制一个按钮时，按钮中的显示文本刚好填充在整个按钮当中，如果动态加长文本的显示长度，就会超出按钮的显示范围，使按钮控件变得不伦不类。我们需要制作控件的自适应，所以不能使用`Rect()`方法固定控件的显示区域，而是需要使用界面布局来制作界面。

使用GUILayout来制作界面，可以很方便地为我们解决上述难题。使用GUI制作界面的时候，需要给每一个控件设定显示区域，如果控件的显示坐标没有计算准确，还会出现控件重叠的情况，而GUILayout无须设定显示区域，系统会自动帮我们计算控件的显示区域，并且保证它们不会重叠。

注意 *之前介绍的大部分GUI控件都可以使用GUILayout进行绘制。*

下面我们将通过一个实例让读者进一步熟悉GUI与GUILayout之间的区别。本例使用GUI与GUILayout分别制作两个按钮，然后动态修改按钮中的文字，看看这两个按钮有什么不同的变化，具体代码如代码清单3-11所示。

代码清单3-11 Script_03_11.js文件

```
var addStr : String = "添加测试字符串";
function OnGUI()
{
    //普通GUI按钮
   if(GUI.Button (Rect (50,50,100,30), addStr))
   {
        addStr +=addStr;
    }
    //界面布局按钮
   if(GUILayout.Button (addStr))
   {
        addStr +=addStr;
    }

}
```

在上述代码中，点击任意按钮，按钮中的文本长度将增加。如图3-16所示，正常情况下两个按钮中的内容与按钮的布局完美适应。

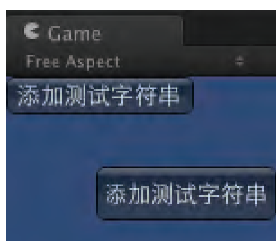


图3-16 正常按钮

点击按钮来增加文本内容，效果如图3-17所示，显然使用GUILayout绘制的按钮按照字符串长度调整了按钮长度，而使用GUI绘制的按钮未能进行自适应调整。由于按钮中的文字增长，而按钮的长度不变，所以按钮上的文字被挤在了中间。由此可以看出，使用GUI绘制的控件非常不灵活。

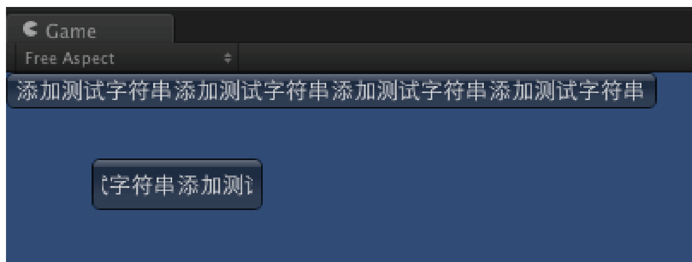


图3-17 自适应按钮

3.2.2　GUILayoutOption界面布局设置

GUILayout是活动的布局，不需要使用`Rect()`方法设定控件的显示区域，而是有一套更为强大的方式来设定控件的显示区域，那就是GUILayoutOption，它可以直接设置某控件的宽度与高度等相关参数。

使用GUILayoutOption设置界面布局时，它无须考虑控件与控件的坐标是否会重叠，GUILayout会自动帮我们把所有控件以线性的排列方式显示在屏幕当中。GUILayoutOption是以一个数组的形式存储设置信息的，比如同时设定某个控件的宽度与高度。

下面我们先简单列举一下GUILayoutOption可填充区域的参数。

- `GUILayout.Width()`：设置布局宽度。
- `GUILayout.Height()`：设置布局高度。
- `GUILayout.MinWidth()`：设置布局最小宽度。
- `GUILayout.MinHeight()`：设置布局最小高度。
- `GUILayout.MaxWidth()`：设置布局最大宽度。
- `GUILayout.MaxHeight()`：设置布局最大高度。
- `GUILayout.ExpandWidth()`：设置布局整体宽度。
- `GUILayout.ExpandHeight()`：设置布局整体高度。

如图3-18所示，本例把不同的界面布局设置应用于5个按钮当中，而无须设置按钮显示的坐标区域，它们之间也不会出现重叠的现象，默认将以垂直线性布局的形式展现，具体代码如代码清单3-12所示。

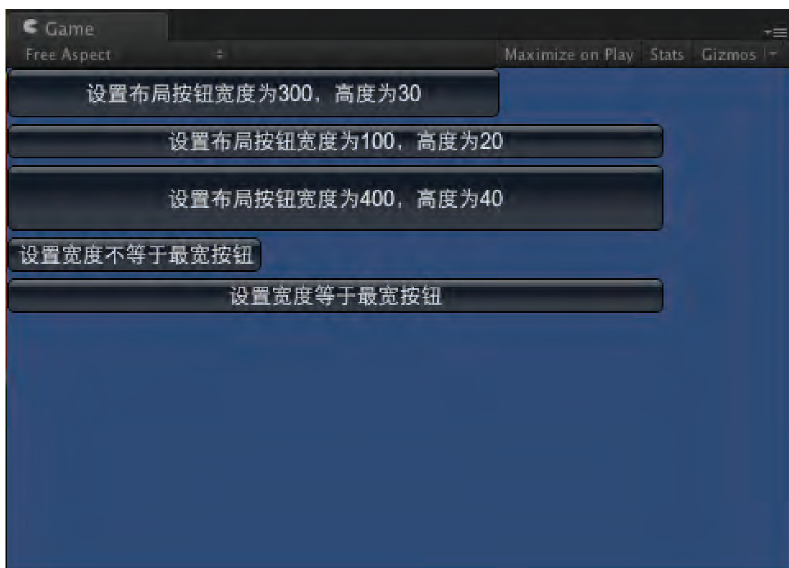


图3-18　按钮的宽度

代码清单3-12　Script_03_12.js文件

```
function OnGUI()
{

    GUILayout.Button ("设置布局按钮宽度为300,高度为30",
        GUILayout.Width(300),GUILayout.Height(30));
    GUILayout.Button ("设置布局按钮宽度为100,高度为20",
        GUILayout.MinWidth(100),GUILayout.MinHeight(20));
    GUILayout.Button ("设置布局按钮宽度为400,高度为40",
        GUILayout.MaxWidth(400),GUILayout.MaxHeight(40));
    GUILayout.Button("设置宽度不等于最宽按钮",GUILayout.ExpandWidth(false));
    GUILayout.Button("设置宽度等于最宽按钮", GUILayout.ExpandWidth(true));

}
```


3.2.3　线性布局

线性布局是以线性连续排列的形式将GUI控件有规律地显示在屏幕中，共分为两种：一种为水平线性布局，另一种为垂直线性布局。默认的界面是以垂直线性布局的方式来排列。

创建水平线性布局时，首先需要使用`BeginHorizontal()`方法，然后将控件添加至线性布局当中，最后使用`EndHorizontal()`方法来结束当前线性布局。而如果使用垂直线性布局，则需要使用`BeginVertical()`方法与`EndVertical()`方法。

无论是水平线性布局还是垂直线性布局，都可以使用嵌套的形式来制作游戏界面，也就是说，父类布局中可以继续嵌套一个子类布局，子类布局完全受父类布局的限制。善用布局之间的嵌套，可以方便我们制作更为复杂的游戏界面。

如图3-19所示，本例分别以垂直线性布局与水平线性布局的形式将一些基本控件显示在屏幕当中，具体代码如代码清单3-13所示。



图3-19　线性布局

代码清单3-13　Script_03_13.js文件

```
//贴图
var texture : Texture2D;
function OnGUI()
{
    //开始水平线性布局
    GUILayout.BeginHorizontal();
    GUILayout.Box("开始水平布局");
    GUILayout.Button ("按钮");
    GUILayout.Label("文本框");
    GUILayout.TextField("输入框");
    GUILayout.Box(texture);
    //结束水平线性布局
    GUILayout.EndHorizontal();

    //开始垂直线性布局
    GUILayout.BeginVertical();
    GUILayout.Box("开始垂直布局");
    GUILayout.Button ("按钮");
    GUILayout.Label("文本框");
    GUILayout.TextField("输入框");
    GUILayout.Box(texture);
    //结束垂直线性布局
    GUILayout.EndVertical();
}
```

在上述代码中，我们先创建了一个水平线性布局，然后又创建了一个垂直线性布局，并且在两种布局中写入相同的基本GUI控件。这两个布局属于平级的关系。由于默认情况下以垂直线性布局排列，所以这两个子类布局将以垂直的的形式排列。

3.2.4　控件偏移

布局与布局之间都是以一种线性方式紧密排列的，无法直接修改布局当中两个相连控件的距离，为了解决这个问题，就需要使用控件偏移。

在布局中，使用`Space()`方法可以设置控件之间的偏移量。如图3-20所示，本例使用布局嵌套的形式将两个垂直线性布局嵌套至水平线性布局当中，分别实现Box控件之间的水平偏移与垂直偏移，具体代码见代码清单3-14所示。

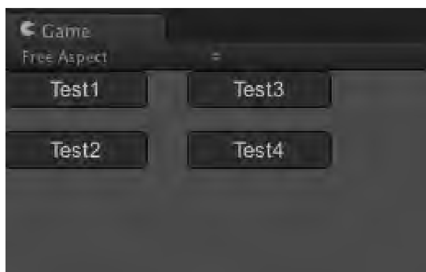


图3-20　布局偏移示例效果图

代码清单3-14　Script_03_14.js文件

```
function OnGUI()
{

    //开始一个显示区域
    GUILayout.BeginArea(Rect(0,0,200,60));

    //开始最外层横向布局
    GUILayout.BeginHorizontal();
    //嵌套一个纵向布局
    GUILayout.BeginVertical();
    GUILayout.Box("Test1");

    //两个Box控件中间偏移10像素
    GUILayout.Space(10);
    GUILayout.Box("Test2");
    //结束嵌套的纵向布局
    GUILayout.EndVertical();
    //两个纵向布局中间偏移20像素
    GUILayout.Space(20);
    //嵌套一个纵向布局
    GUILayout.BeginVertical();
    GUILayout.Box("Test3");

    //两个Box控件中间偏移10像素
    GUILayout.Space(10);
    GUILayout.Box("Test4");
    //结束嵌套的纵向局部
    GUILayout.EndVertical();

    //结束最外层横向布局
    GUILayout.EndHorizontal();
    //结束显示区域
    GUILayout.EndArea();

}
```

使用布局偏移时，需要在布局中确认两个控件并在这两个控件之间进行偏移。在第一个控件绘制完毕后，紧接着使用代码`GUILayout.Space()`进行偏移，其中`Space()`方法的参数为偏移的距离。

3.2.5　对齐方式

一般情况下，游戏窗口都是矩形的，而矩形由4个角组成：左上角、右上角、左下角和右下角。如果要使控件分别对齐在这4个角当中，就需要使用`FlexibleSpace()`对其做偏移了。`FlexibleSpace()`的原理是将两个控件完全左右或上下对齐在显示区域当中。它是一个非常重要的方法，在不同分辨率下可以直接确定偏移的位置，并且不会超出显示范围。

如图3-21所示，本例设置了一个全屏大小的显示区域，分别在这个区域的左上、右上、左下和右下4个角当中填充一个Box控件，具体代码如代码清单3-15所示。

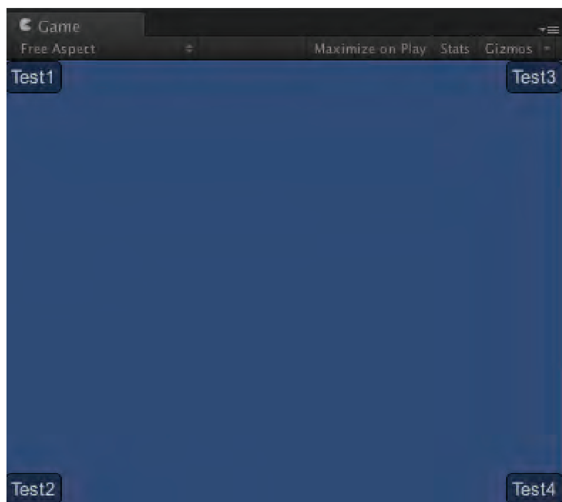


图3-21　对齐方式

代码清单3-15　Script_03_15.js文件

```
function OnGUI()
{
    //开始一个显示区域
    GUILayout.BeginArea (Rect (0,0,Screen.width,Screen.height));

    //开始最外层横向布局
    GUILayout.BeginHorizontal();
    //嵌套一个纵向布局
    GUILayout.BeginVertical();

    GUILayout.Box("Test1");
    //两个Box控件上下对齐
    GUILayout.FlexibleSpace();
    GUILayout.Box("Test2");
    //结束嵌套的纵向局部
    GUILayout.EndVertical();

    //布局之间左右对齐
    GUILayout.FlexibleSpace();

    //嵌套一个纵向布局
    GUILayout.BeginVertical ();

    GUILayout.Box("Test3");
    //两个Box控件上下对齐
    GUILayout.FlexibleSpace();
    GUILayout.Box("Test4");
    //结束嵌套的纵向布局
    GUILayout.EndVertical();

    //结束最外层横向布局
```

```
        GUILayout.EndHorizontal();
        //结束显示区域
        GUILayout.EndArea();

    }
```

使用GUILayout.FlexibleSpace()方法设置对齐方式时，它会自动获取当前屏幕的宽或高，确保两个控件相互对齐在屏幕的两端并且不会超出屏幕。

3.2.6　实例——添加与关闭窗口

游戏窗口是可以动态添加与关闭的。本例中，我们将制作一个可以动态添加与关闭的窗口，如图3-22所示。在屏幕中点击“添加新窗口”或“关闭当前窗口”按钮，窗口将执行添加与删除操作。

在程序中，我们将窗口的属性存储在数组链表中（无论是添加窗口还是关闭窗口，实际上就是在维护这个窗口的数组链表），然后使用OnGUI()方法遍历窗口并将窗口绘制在屏幕当中，具体代码如代码清单3-16所示。

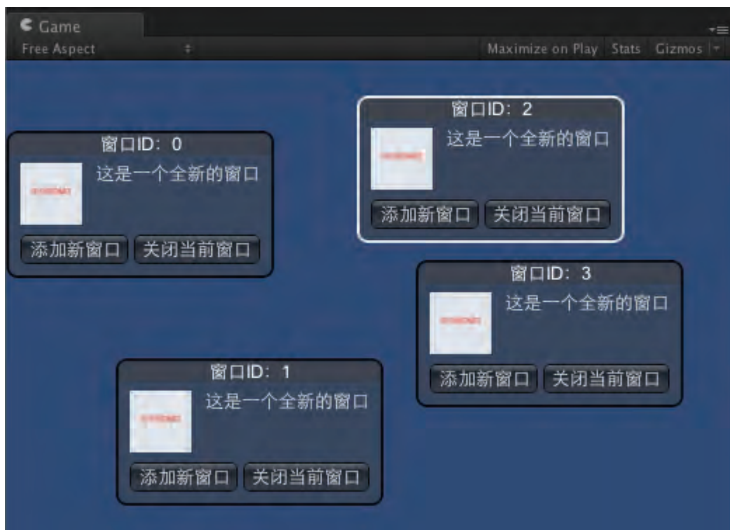


图3-22　添加与关闭窗口示例效果图

代码清单3-16　Script_03_16.js文件

```
//使用ArrayList存储窗口
var winArrayList = new ArrayList();

//图标
var icon : Texture;

function Start()
{
```

```
        //添加一个窗口
        winArrayList.Add(Rect(winArrayList.Count*100,50,150,100));
    }

    function OnGUI()
    {

        //遍历每个窗口，并且加入视图
        var count = winArrayList.Count;
        for(var i = 0; i < count;i++)
        {
            winArrayList[i]= GUILayout.Window (i, winArrayList[i], AddWindow, "窗口ID："+i);

        }

    }

    function AddWindow (windowID:int)
    {

        //开始一个水平布局
        GUILayout.BeginHorizontal();
        //绘制图标
        GUILayout.Label(icon,GUILayout.Width(50),GUILayout.Height(50));
        //绘制文字
        GUILayout.Label("这是一个全新的窗口");
        //关闭水平布局
        GUILayout.EndHorizontal();

        //开始一个水平布局
        GUILayout.BeginHorizontal();

        if (GUILayout.Button ("添加新窗口"))
        {
            //添加窗口
            winArrayList.Add(Rect(winArrayList.Count*100,50,150,100));
        }

        if (GUILayout.Button ("关闭当前窗口"))
         {
            //关闭窗口
            winArrayList.RemoveAt(windowID);
        }
        //关闭水平布局
        GUILayout.EndHorizontal();
        //设置窗口拖动的区域
        GUI.DragWindow(Rect(0,0, Screen.width, Screen.height));

    }
```

在上述代码中，我们需要维护winArrayList这个数组链表，其中将详细记录每一个窗口的信息。点击“添加新窗口”按钮后，将给这个链表添加一条新窗口的数据，点击“关闭当前窗口”按钮，则表示在链表中删除这个窗口数据。

当前链表的长度就是窗口的总数量。在OnGUI()方法中使用for循环去遍历这个链表，在循环中使用GUILayout.Window()方法最终将所有窗口绘制在屏幕当中。

3.2.7　设置字体

在实际开发中，有的游戏比较崇尚张扬与个性，此时系统默认的字体就不会被开发者所青睐，需要考虑修改显示字体。Unity支持所有.ttf的字符集，默认的字体为Arial。

设置字体前，首先需要得到一个.ttf的字符集，获取方式有很多，比如通过网络下载或者复制本机电脑中的字符集，本节中，我们直接复制本机电脑中的字符集。苹果电脑中所有字体的存放路径为“Finder”→“资源库”→“Fonts”。打开文件夹后，发现里面存放了各式各样的字体，选择“华文宋体.ttf”与“华文楷体.ttf”这两种字体并且将其拖放至Unity当前工程资源文件夹下。

在Project视图中选择“Create”→“GUI Skin”菜单项，创建一个GUI皮肤来设置我们的自定义字体，如图3-23所示。GUI Skin的默认字体为Arial。这里我们设置Box的字体为“华文宋体”，设置Button的字体为“华文楷体”，剩下的控件都采用默认字体Arial，具体代码如代码清单3-17所示。

另外，也可以在Project视图中自行创建字体，具体方法为选择“Create”→“Custom Font”菜单项即可。

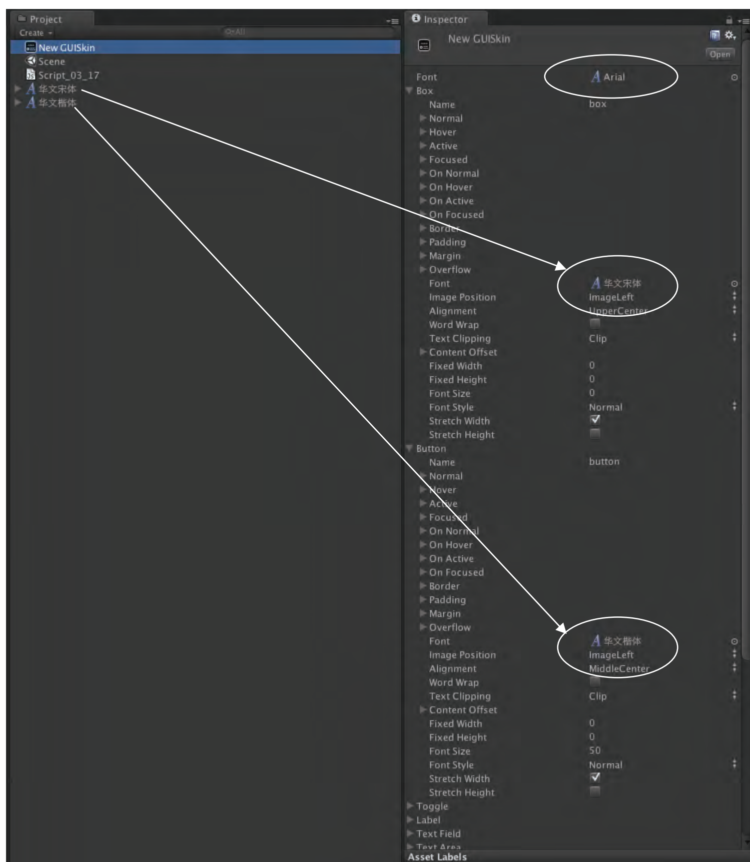


图3-23　自定义字体

代码清单3-17　Script_03_17.js文件

```
//皮肤
var mySkin : GUISkin;

function OnGUI()
{
    //设置皮肤
    GUI.skin = mySkin;
    GUILayout.Box("华文宋体");
    GUILayout.Button("华文楷体");
    GUILayout.Label("默认字体");
}
```

运行游戏后，三种不同的字体已经出现在Game视图当中，如图3-24所示。



图3-24　自定义字体效果图

下面我们来学习如何修改自定义控件中字体的大小。首先在Project视图中选择GUI Skin，此时右侧的Inspector视图中将弹出GUI Skin的相关选项。下面我们尝试修改按钮中文字的大小。首先选择要修改的控件名称为“Button”，然后在下方的“Font Size”输入框中输入50即可，如图3-25所示。

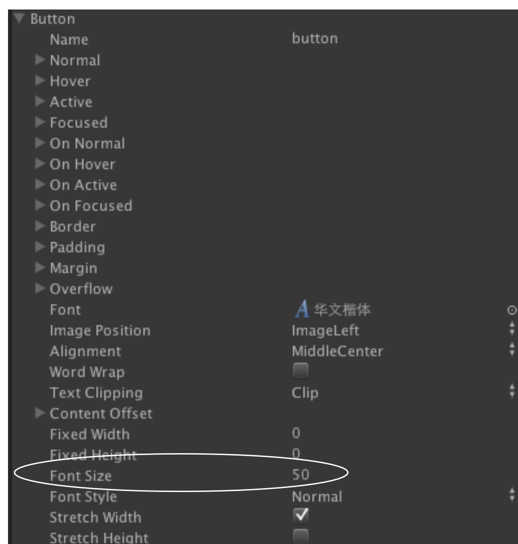


图3-25　设置字体大小

设置完字体大小后，直接运行游戏，即可看到图中按钮的字体已经变大，如图3-26所示。



图3-26　改变字体大小后的效果

3.2.8　显示中文

Unity脚本默认的编码是Western(ISO latin 1)，这种编码是不支持中文的。下面我们分别使用JavaScript与C#脚本在屏幕中绘制含有中文的标签，JavaScript脚本如下所示：

```
function OnGUI()
{
    GUI.Label( Rect(0,10,100,30),"JavaScript中文测试");

}
```

C#脚本如下所示：

```
using UnityEngine;
using System.Collections;

public class GUI_18 : MonoBehaviour
{

    void OnGUI()
    {
        GUI.Label(new Rect(1,50,100,30),"C#中文测试");

    }
}
```

直接运行游戏，默认编码的脚本中的英文正常显示在屏幕中，而中文未能正常显示，而是以“？”的形式呈现，如图3-27所示。

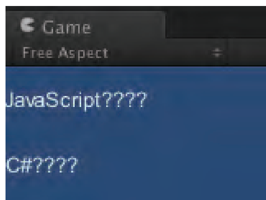


图3-27　中文测试

为了可以正常显示中文，我们需要修改脚本的编码格式。JavaScript需要将脚本编码格式修改为Unicode(UTF-8)，C#需要将脚本编码格式修改为Unicode(UTF-16)。

下面我们打开Unitron应用程序来修改脚本的编码格式。Unitron也是Unity提供的一个脚本编辑器，它安装在Unity根目录中。使用它打开JavaScript或C#脚本后，在导航菜单栏中选择"Text"→"Text Encoding"菜单项（如图3-28所示），选择编码格式后，使用快捷键Command + S保存即可。

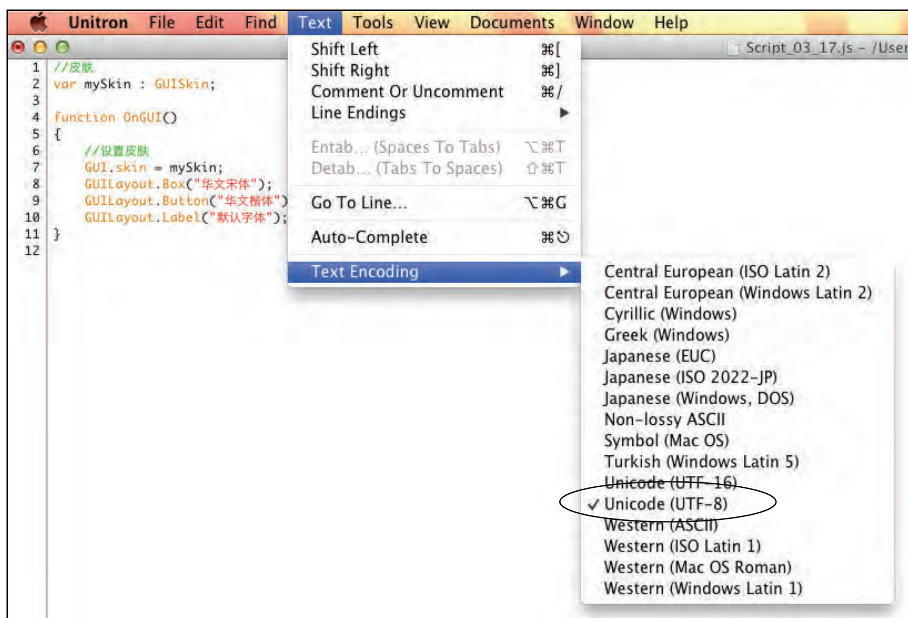


图3-28　修改编码格式

此时运行游戏，可以发现中文正常显示出来了，如图3-29所示。



图3-29　正常显示中文的效果

3.3 2D 贴图与帧动画

2D贴图好比在屏幕中绘制了一张静态图片，其绘制方式有两种，第一种由GUI绘制，第二种是将贴图以材质的形式绘制在游戏对象中。在本节中，我们将着重介绍第一种方式。

帧动画的实现原理就是使用若干张静态图片以一定的时间一帧一帧地在屏幕中切换播放，好比在屏幕中预先设定一个显示动画的区域，然后将图片在这个显示区域中频繁切换播放。由于绘制的图片有规律地切换播放，给人们带来了视觉的假象，感觉就像播放动画一样。

3.3.1 绘制贴图

要在屏幕中绘制一张静态贴图，需要使用GUI.DrawTexture()方法，该方法可设定图片的显示位置、缩放比例和渲染混合等，该方法的原型如下：

```
GUI.DrawTexture(Rect(110,10,120,120), texSingle, ScaleMode.StretchToFill, true, 0);
```

其中第一个参数表示图片的绘制区域，第二个参数表示绘制图片的对象，第三个参数表示图片缩放模式，第四个参数表示是否开启图片混合模式，第五个参数表示图片缩放宽高的比例。

在Project视图中将需要加载的图片存储在根目录“Resources”中，如图3-30所示。需要说明的是，一定要将加载的图片保存在“Resources”文件夹中，否则程序将无法识别。

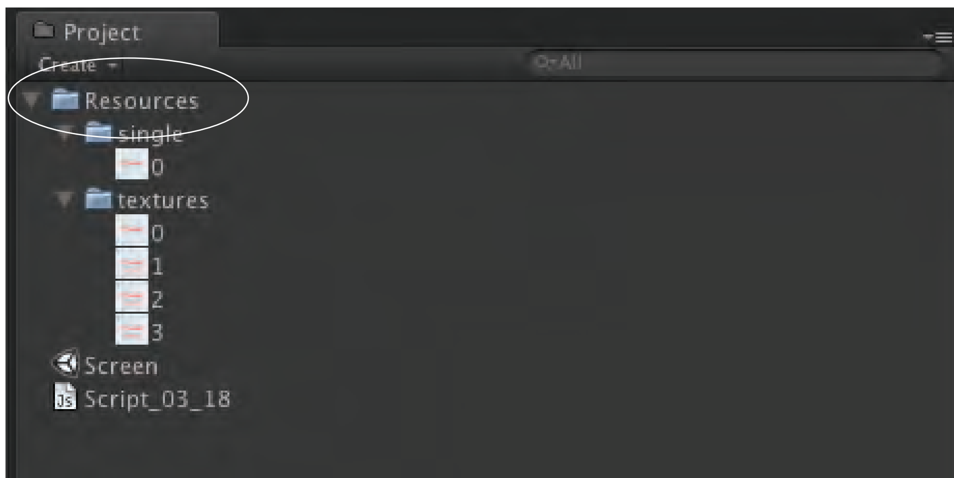


图3-30 加载图片

其中Resources.Load()方法和Resources.LoadAll()方法的参数均为资源文件夹的完整路径，只不过前者返回的是读取的资源对象，后者返回的是资源对象的数组。

如图3-31所示，本例分别读取了单个图片与多个图片，并且将加载的图片绘制在屏幕当中，具体代码如代码清单3-18所示。

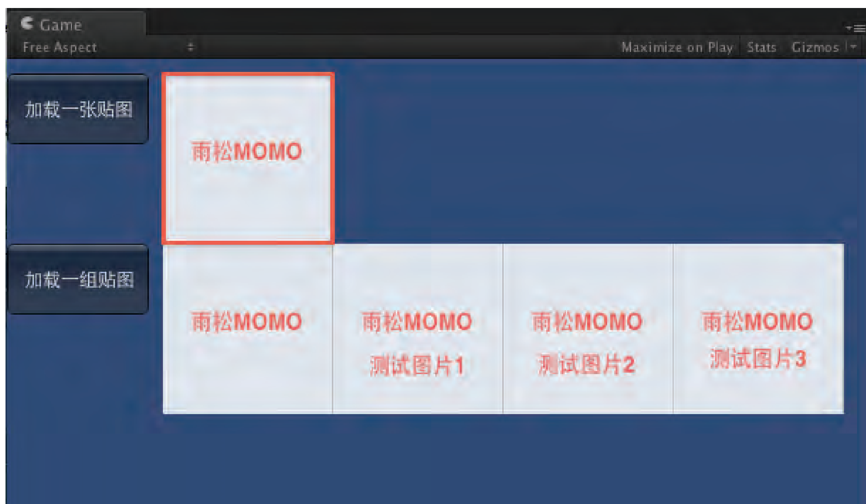


图3-31　加载图片

代码清单3-18　Script_03_18.js文件

```
//贴图
private var texSingle : Texture2D;
//贴图数组
private var texAll : Object[] ;


function OnGUI()
{
    if(GUI.Button(Rect(0,10,100,50),"加载一张贴图"))
    {
        if(texSingle == null)
        {
            //加载贴图
            texSingle =  Resources.Load("single/0");
        }
    }

    if(GUI.Button(Rect(0,130,100,50),"加载一组贴图"))
    {
        if(texAll == null)
        {
            //加载所有贴图
            texAll = Resources.LoadAll("Textures");
        }
    }

    //绘制贴图
    if(texSingle != null)
    {
```

```
            //绘制一张贴图
            GUI.DrawTexture(Rect(110,10,120,120), texSingle, ScaleMode.StretchToFill,
                true, 0);
        }

        if(texAll != null)
        {
            for (var i = 0; i < texAll.length; i++) {
                //绘制贴图
                GUI.DrawTexture(Rect(110 + i * 120,130,120,120), texAll[i],
                    ScaleMode.StretchToFill, true, 0);
            }
        }

    }
```


3.3.2　绘制动画

本节中我们开始学习帧动画的绘制。首先需要一组帧动画的资源，这里我们选择一套2D人物四宫格行走图，如图3-32所示。在绘制帧动画之前，我们需要学习帧动画的绘制原理：首先需要在屏幕中设定一个显示区域，然后将动画中的每一帧图片按照固定的时间在这个区域按中按顺序切换，继而实现动画的播放。

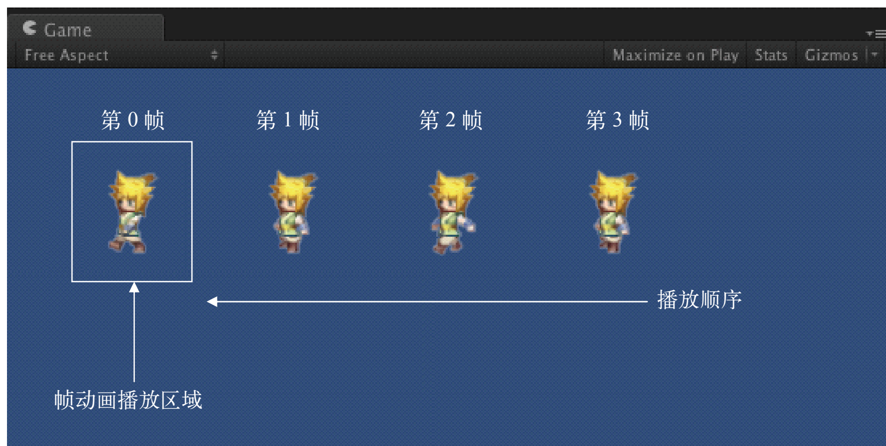


图3-32　加载图片

这里我们使用程序将动画资源存储在动画数组当中，然后设定动画的刷新时间，每次刷新动画时将在原有显示区域中绘制下一帧图片，到了最后一帧则从第一帧重新开始，依次类推。

如图3-33所示，人物行走循环动画已经在屏幕中开始播放，并且将播放帧数的序列号以标签的形式绘制在屏幕当中，具体代码如代码清单3-19所示。

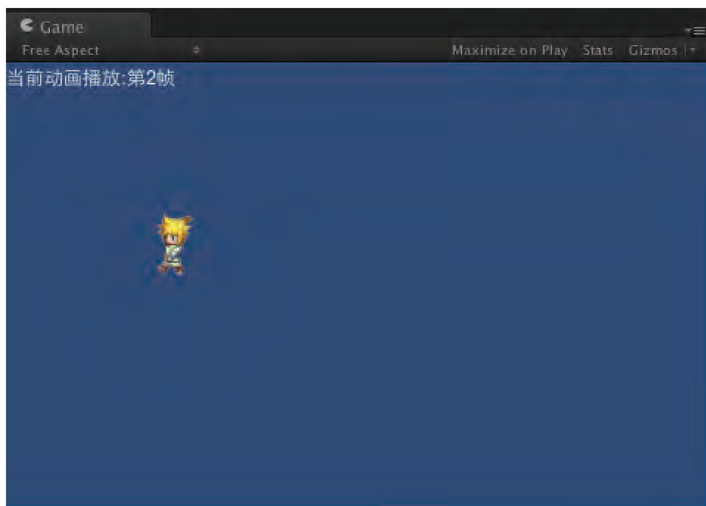


图3-33　播放帧动画

代码清单3-19　Script_03_19.js文件

```
//动画数组
private var anim : Object[] ;
//帧序列
private var nowFram : int;
//动画帧的总数
private var mFrameCount : int;
//限制一秒多少帧
private var fps : float = 15;
//限制帧的时间
private var time : float = 0;

function Start()
{
    //得到帧动画中的所有图片资源
    anim = Resources.LoadAll("animation");
    //得到该动画共有多少帧
    mFrameCount = anim.Length;
}

function OnGUI()
{
    //绘制帧动画
    DrawAnimation(anim,Rect(100,100,32,48));
}

function  DrawAnimation(tex:Object[] , rect : Rect)
{
    //绘制动画信息
    GUILayout.Label("当前动画播放：第"+nowFram+"帧");
```

```
        //绘制当前帧
        GUI.DrawTexture(rect, tex[nowFram], ScaleMode.StretchToFill, true, 0);
        //计算限制帧时间
        time += Time.deltaTime;
        //超过限制帧则切换图片
        if(time >= 1.0 / fps){
            //帧序列切换
            nowFram++;
            //限制帧清空
            time = 0;
            //超过帧动画总数，从第0帧开始
            if(nowFram >= mFrameCount)
            {
                nowFram = 0;
            }
        }
    }
```

在本示例中，我们将播放动画的功能封装在DrawAnimation()方法中，该方法的第一个参数表示动画资源数组对象，第二个参数表示动画的显示区域。

3.3.3　实例——人物移动

结合2D帧动画的绘制原理，本节我们将制作一个游戏实例，效果如图3-34所示。在屏幕中共绘制了4个按钮，通过点击这4个按钮来控制主角的移动，并且播放主角在对应方向上行走的动画，具体代码如代码清单3-20所示。



图3-34　行走示例

3

代码清单3-20　Script_03_20.js文件

```
//动画数组
private var animUp : Object[] ;
private var animDown : Object[] ;
private var animLeft : Object[] ;
private var animRight : Object[] ;
//地图贴图
private var map : Texture2D;
//当前人物动画
private var tex : Object[];
//人物的x坐标
private var x:int;
//人物的y坐标
private var y:int;
//帧序列
private var nowFram : int;
//动画帧的总数
private var mFrameCount : int;
//限制一秒多少帧
private var fps : float = 10;
//限制帧的时间
private var time : float = 0;

function Start()
{

    //得到帧动画中的所有图片资源
    animUp = Resources.LoadAll("up");
    animDown = Resources.LoadAll("down");
    animLeft = Resources.LoadAll("left");
    animRight = Resources.LoadAll("right");
    //得到地图资源
    map = Resources.Load("map/map");
    //设置默认动画
    tex  = animUp;
}

function OnGUI()
{
    //绘制贴图
    GUI.DrawTexture(Rect(0,0,Screen.width,Screen.height), map,
        ScaleMode.StretchToFill, true, 0);

    //绘制帧动画
    DrawAnimation(tex,Rect(x,y,32,48));

    //点击按钮移动人物
    if(GUILayout.RepeatButton("向上"))
    {
        y-=2;
        tex = animUp;
    }
```

```
        if(GUILayout.RepeatButton("向下"))
        {
            y+=2;
            tex = animDown;
        }
        if(GUILayout.RepeatButton("向左"))
        {
            x-=2;
            tex = animLeft;
        }
        if(GUILayout.RepeatButton("向右"))
        {
            x+=2;
            tex = animRight;
        }
    }

    function  DrawAnimation(tex : Object[] , rect : Rect)
    {
        //绘制当前帧
        GUI.DrawTexture(rect, tex[nowFram], ScaleMode.StretchToFill, true, 0);
        //计算限制帧时间
        time += Time.deltaTime;
        //超过限制帧则切换图片
        if(time >= 1.0 / fps){
            //帧序列切换
            nowFram++;
            //限制帧清空
            time = 0;
            //超过帧动画总数，从第0帧开始
            if(nowFram >= tex.Length)
            {
                nowFram = 0;
            }
        }
    }
```

在本例中，程序需要监听用户触发的按钮来切换动画方向，比如当用户点击“向上”按钮时，将播放主角向上行走的动画。

在上述代码中，我们使用x、y全局变量来记录当前主角的坐标，上下行走为加减y坐标，左右行走为加减x坐标，最后根据主角的x、y坐标来绘制当前动画在屏幕中的区域，从而实现控制主角向4个方向行走。

3.3.4　实例——用 Unity 开发 2D 游戏

在上一节中，我们学习了如何使用GUI来实现2D动画的播放。实际上，在开发2D游戏时，不能使用GUI来实现。GUI的渲染效率比较低，并且也无法使用Unity引擎中独具特色的功能，比如物理引擎、粒子系统和特效等。大家想一想2D游戏与3D游戏的区别，3D游戏就是将z轴加入到游戏当中，如果不去旋转摄像机，让它直接照射x轴与y轴，那么它就与2D游戏一样了。

在Unity中制作2D游戏的方法如下：首先在游戏世界中创建一个面对象，它就是2D游戏的背景地图，并且让摄像机直对着它；接着在背景地图上继续绘制图层，它依然由面对象组成，比如主角、敌人和道具等；最后动态更新每个面对象身上的材质贴图即可实现帧动画。如图3-35所示，我们在Scene视图中可以清晰地看到3个面对象，游戏的背景面与两个人物的贴图面。本例代码如代码清单3-21所示。

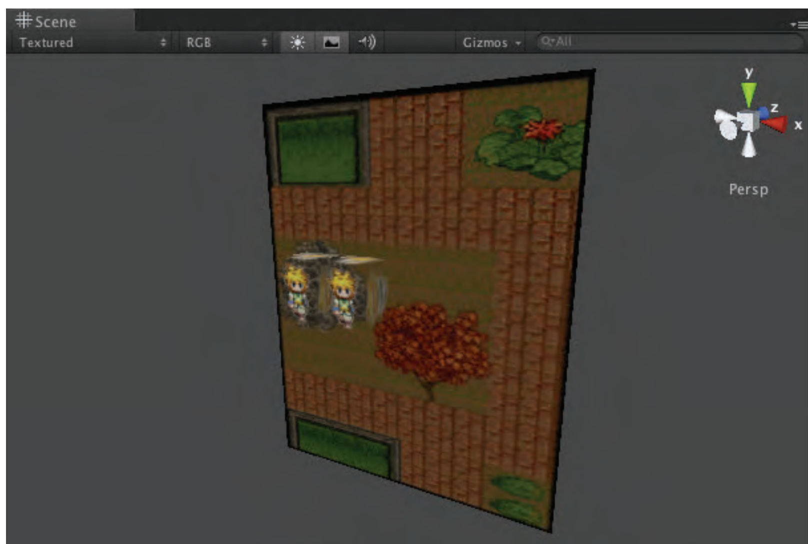


图3-35 场景示例

代码清单3-21 Script_03_21.js文件

```
//主角对象
private var hero : GameObject;

//按键是否被按下
private var keyUp : boolean;
private var keyDown : boolean;
private var keyLeft : boolean;
private var keyRight : boolean;

//记录当前时间
private var time : float;
//限制一秒多少帧
private var fps : float = 10;
//帧序列
private var nowFram : int;
//动画数组
private var animUp : Object[] ;
private var animDown : Object[] ;
private var animLeft : Object[] ;
```

```
private var animRight : Object[];
private var nowAnim : Object[];
private var backAnim : Object[];

function Start()
{
    //得到主角对象
    hero = GameObject.Find("hero");
    //得到上下左右四组动画
    animUp = Resources.LoadAll("up");
    animDown = Resources.LoadAll("down");
    animLeft = Resources.LoadAll("left");
    animRight = Resources.LoadAll("right");
    nowAnim = animDown;
    backAnim = animDown;
}

function OnGUI()
{
    //控制主角移动的按钮
    keyUp = GUILayout.RepeatButton("向上");
    keyDown = GUILayout.RepeatButton("向下");
    keyLeft = GUILayout.RepeatButton("向左");
    keyRight = GUILayout.RepeatButton("向右");

}

function FixedUpdate()
{

    if(keyUp)
    {
        //向上移动
        SetAnimation(animUp);
        hero.transform.Translate(-Vector3.forward * 0.001f);

    }

    if(keyDown)
    {
        //向下移动
        SetAnimation(animDown);
        hero.transform.Translate(Vector3.forward * 0.001f);

    }
    if(keyLeft)
    {
        //向左移动
        SetAnimation(animLeft);
        hero.transform.Translate(Vector3.right * 0.001f);
```

```
    }

    if(keyRight)
    {
        //向右移动
        SetAnimation(animRight);
        hero.transform.Translate(-Vector3.right * 0.001f);

    }
    //播放动画
    DrawAnimation(nowAnim);
}

function  DrawAnimation(tex : Object[])
{

    //计算限制帧的时间
    time += Time.deltaTime;
    //超过限制帧的切换贴图
    if(time >= 1.0 / fps){
        //帧序列切换
        nowFram++;
        //限制帧清空
        time = 0;
        //超过帧动画总数时，从第0帧开始
        if(nowFram >= tex.Length)
        {
            nowFram = 0;
        }
    }
    //将对应的贴图赋予主角对象
    hero.renderer.material.mainTexture = tex[nowFram];
}

function SetAnimation(tex : Object[])
{
    //设置播放动画
    nowAnim = tex;
    if(!backAnim.Equals(nowAnim))
    {

        nowFram = 0;
        backAnim = nowAnim;
    }

}
```

因为默认的材质会将贴图中透明的部分渲染成白色，所以主角原本透明的背景被渲染成了白色，如图3-36所示。



图3-36　背景贴图

为了让图片的背景呈透明状，需要设置一下材质的渲染通道。如图3-37所示，在Project视图中选择主角的材质资源，接着在右侧的Inspector视图中找到Shader下拉列表，然后从中选择“Transparent”→“Diffuse”菜单项即可。

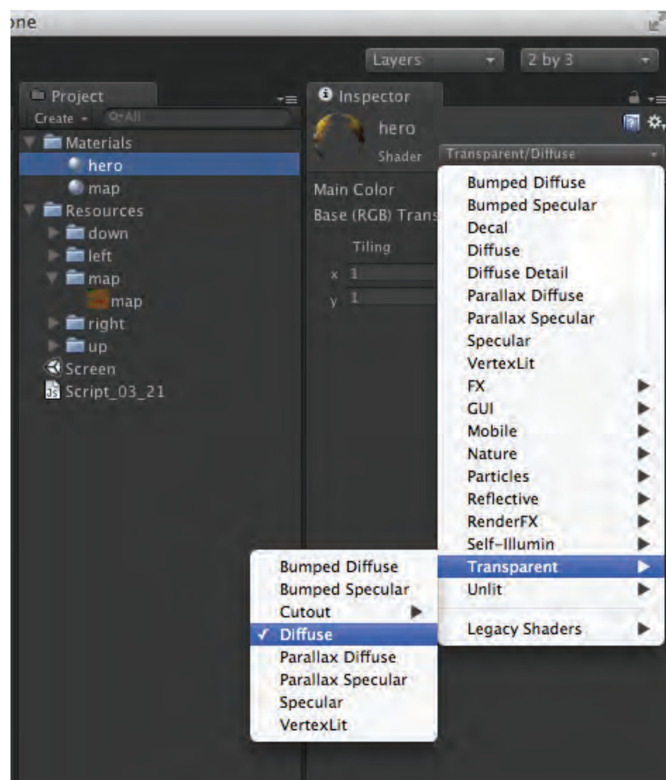


图3-37　处理材质

此外，游戏对象是可以添加游戏组件的，为了让主角与其他对象发生碰撞，可以给主角绑定一个刚体组件，操作方法是在Hierarchy视图中选择主角对象，接着在Unity导航菜单栏中选择“Component”→“Physics”→“Rigidbody”菜单项即可。如图3-38所示，移动主角，当它碰撞至其他对象时立即停了下来。



图3-38　游戏碰撞

3.4　游戏实例——游戏主菜单

在本节中，我们将制作一个游戏的主菜单。一般情况下，游戏主菜单包括游戏标题、按钮和动画特效等。如图3-39所示，进入游戏后，Game视图的正上方为游戏的标题，它以2D贴图方式呈现。下方为一些功能性的按钮。图中的小人为主菜单中的特殊动画，小人始终从右向左来移动，如果小人超过左边边界后，他就再回到原位，保持动画一直向左移动，具体代码如代码清单3-22所示。

代码清单3-22　Script_03_22.js文件

```
//背景贴图
private var bg : Texture2D;
//标题贴图
private var title : Texture2D;
//动画数组
private var tex : Object[];
//动画x坐标
```

```
private var x:int;
//动画y坐标
private var y:int;
//帧序列
private var nowFram : int;
//动画帧的总数
private var mFrameCount : int;
//限制一秒多少帧
private var fps : float = 5;
//限制帧的时间
private var time : float = 0;

function Start()
{
    //载入资源
    bg = Resources.Load("bg");
    title = Resources.Load("title");
    tex =  Resources.LoadAll("anim");
    //初始化动画坐标
    x = Screen.width;
    y = 200;
}

function OnGUI()
{
    //绘制贴图
    GUI.DrawTexture(Rect(0,0,Screen.width,Screen.height), bg,
        ScaleMode.StretchToFill, true, 0);
    //绘制标题
    GUI.DrawTexture(Rect((Screen.width - title.width)>>1,30,title.width,title.
        height), title, ScaleMode.StretchToFill, true, 0);

    //绘制帧动画
    DrawAnimation(tex,Rect(x,y,40,60));
    //动画越界监测
    x --;
    if(x <-42)
    {
        x =480;

    }
    //绘制按钮
    GUI.Button(Rect(230,200,100,30),"开始游戏");
    GUI.Button(Rect(230,240,100,30),"读取进度");
    GUI.Button(Rect(230,280,100,30),"关于游戏");
    GUI.Button(Rect(230,320,100,30),"退出游戏");
}


function  DrawAnimation(tex : Object[] , rect : Rect)
{
    //绘制当前帧
    GUI.DrawTexture(rect, tex[nowFram], ScaleMode.StretchToFill, true, 0);
    //计算限制帧时间
```

3

```
        time += Time.deltaTime;
        //超过限制帧则切换图片
        if(time >= 1.0 / fps){
            //帧序列切换
            nowFram++;
            //限制帧清空
            time = 0;
            //超过帧动画总数，从第0帧开始
            if(nowFram >= tex.Length)
            {
                nowFram = 0;
            }
        }
    }
```



图3-39　游戏菜单

在本例中，我们结合之前的学习制作了一个简单的游戏菜单，它包括背景图片、游戏标题和游戏按钮。此外，为了让游戏菜单更加丰富好看，我们还将帧动画添加了进去。

3.5　本章小结

本章首先介绍了Unity中GUI界面的相关组件以及自定义皮肤的实现方式，其中每个GUI高级组件都配备了一个小例子供读者学习；然后介绍了GUI与GUILayout之间的区别，以及如何使用GUILayout布局来制作界面；接着介绍了如何使用GUI绘制2D贴图与动画，以及如何制作控制主角移动的游戏实例；最后通过制作一个游戏主界面，回顾了前面所学游戏界面相关的内容。请大家认真阅读本章内容，打好游戏界面设计的基础，为后面深入学习做好准备。

第4章

Unity游戏脚本

Unity游戏脚本在整个游戏开发中可以说是关键要素，游戏对象之间任何逻辑的判断都需要通过脚本来完成。如果说游戏贴图、模型资源的好坏决定一个游戏的视觉品味，那么脚本将直接决定这个游戏的内在质量，决定这个游戏好玩与否。游戏脚本与其他游戏组件用法相同，必须绑定在游戏对象中才能执行它的生命周期。

Unity一共支持3种语言来编写脚本，分别是JavaScript、C#和Boo，这3种语言不分好坏，用哪一种来编写都可以达到同样目的。从编程技巧与难度上来讲，JavaScript更容易上手一些，建议初学者使用JavaScript进行入门阶段学习，但是进阶阶段推荐使用C#语言来编写脚本，因为C#语言在编程思想上更符合Unity引擎的原理。由于与传统语言相比，Boo语言的语法更为怪异，所以开发中几乎不会用到它。Boo语言的语法非常接近Python，有兴趣的读者可以自行研究。

在本书中，在入门阶段我们将使用JavaScript语言来讲解，介绍进阶方面的内容时，将以C#语言进行讲解。本章先以JavaScript语言来学习如何编写Unity游戏脚本。

4.1 MonoDevelop 脚本编辑器

Unity可部署在Windows与Mac OS X两种操作系统下，所以Unity需要一个跨平台的脚本编辑器。MonoDevelop脚本编辑器并不是Unity公司所研发的，它是一个开源项目，任何人或公司都可以使用。由于该编辑器具有强大的跨平台功能，并且使用起来非常方便，所以很快被Unity公司作为核心脚本开发环境来使用。

4.1.1 编辑器简介

MonoDevelop可以同时使用JavaScript、C#和Boo三种语言来编辑脚本（书中主要介绍JavaScript和C#这两种语言的用法），并且已经集成在Unity安装包中。

安装Unity时，会默认安装MonoDevelop脚本编辑器。如图4-1所示，MonoDevelop就在Unity的根目录下。

打开Unity时，首先需要修改一下默认脚本的打开方式，具体操作如下：在Unity编辑器界面上方的菜单栏中点击“Unity”→“Preferences”（首选项）菜单项，打开“Unity Preferences”界面，选择左侧的“External Tools”菜单项，然后在“External Script Editor”下拉列表中选择“Browse…”选项（如图4-2所示），然后再在“Finder”中寻找游戏脚本的默认打开方式，这里我

们选择MonoDevelop。



图4-1 Unit编辑器路径

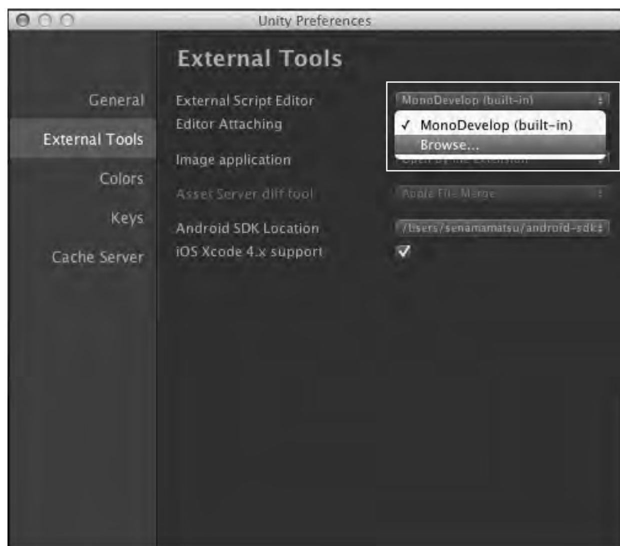


图4-2 设置脚本打开方式

4.1.2 调试

MonoDevelop脚本编辑器不仅可以编写游戏脚本，还提供了调试功能，可以方便开发者调试自己的游戏程序。调试脚本时，需要设置一些调试参数，下面简要介绍一下。

打开MonoDevelop脚本编辑器,点击左上角的“MonoDevelop”→“Preferences”（首选项）菜单项，在打开的“选项”窗口中将“Editor Location”下拉列表设置为调试默认启动的应用程序，所以这里将其设置为Unity.app，如图4-3所示。“Launch Unity automatically”复选框表示是否启动调试,这里必须选中它，否则无法调试。“Build project in MonoDevelop”复选框表示是否在脚本编辑器

中构建项目。设置完毕后，点击“确定”按钮，即可完成参数设置。

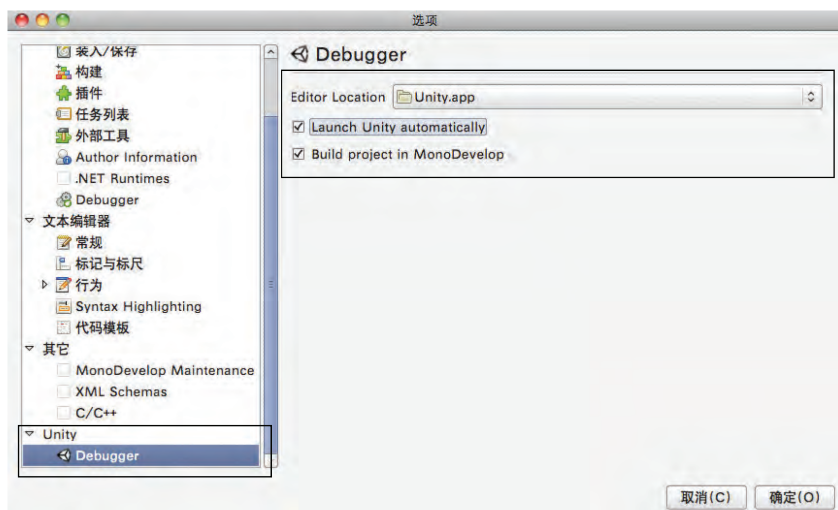


图4-3　编辑器设置

此外，在调试之前，我们还需要创建脚本文件。首先打开Unity，然后在Project视图中选择“Create”→“JavaScript”菜单项，创建一条JavaScript游戏脚本，最后将下面这段简单的代码写入脚本中：

```
function Start()
{
    var i = 100;
    var j = i;
    Debug.Log(j);
}
```

上述代码所执行的内容是简单的赋值操作。代码编写完毕后，我们先将这条脚本绑定在摄像机对象中，然后在这条脚本中添加程序断点。添加断点的方式是，在代码中将光标放置在需要添加断点的那一行，然后在MonoDevelop脚本编辑器的导航菜单栏中选择“运行”→“切换断点”菜单项（如图4-4所示），也可在需要添加断点代码的左侧单击鼠标左键。

接着就可以调试这段程序了。首先在MonoDevelop菜单栏中点击“运行”→“Run With”→“Unity Debugger”菜单项（如图4-5所示），开启MonoDevelop的程序调试模式，此时程序将自动唤醒当前脚本对应的Unity工程，然后在Unity编辑器上方的工具栏中点击“运行游戏”按钮，程序将正式开始调试。

如图4-6所示，程序已经停在了添加断点的那一行。在顶部的工具栏中，有4个比较重要的按钮，依次是停止调试、逐过程单步调试、逐语句单步调试和跳出当前。左边的“解决方案”为当前脚本工程的资源路径。将鼠标放在需要查看的变量之上，即可弹出属性小窗口，以方便查看程序运行中某一个变量的数值。

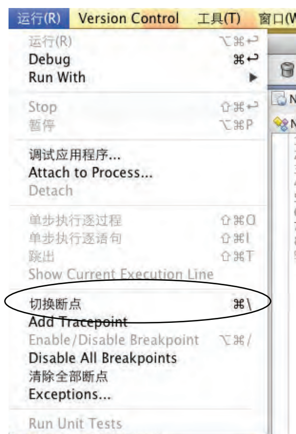


图4-4　“运行”菜单



图4-5　运行调试的菜单

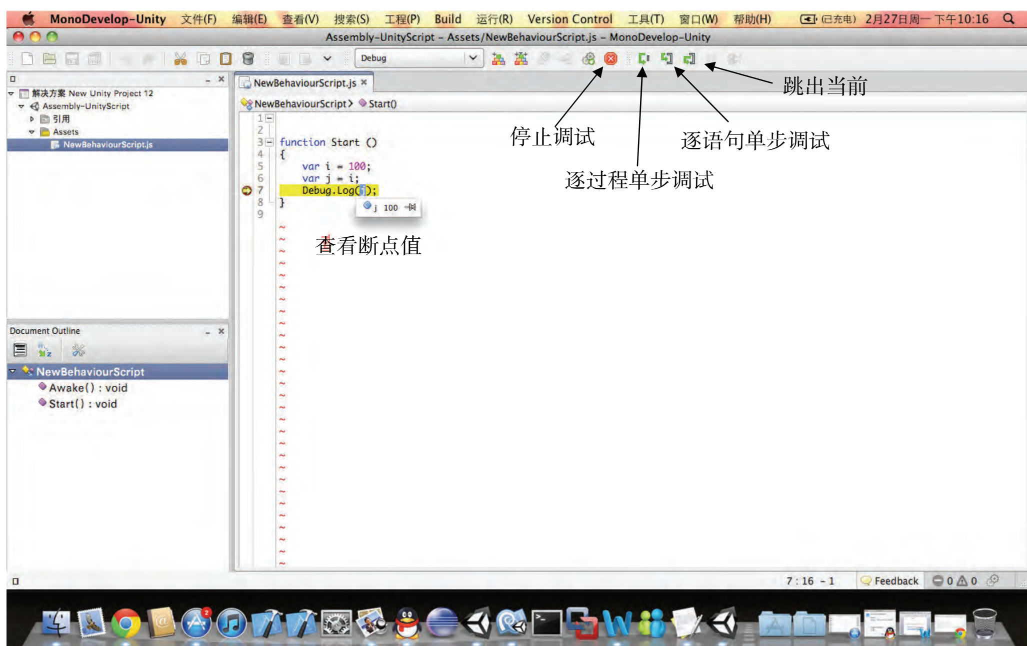


图4-6　调试界面

4.2 Unity 脚本的生命周期

Unity脚本从唤醒到销毁有着一套比较完善的生命周期，添加任何脚本都必须遵守自身生命周期法则。下面介绍一下生命周期中由系统自身调用的几个比较重要的方法。

- `function Update(){}`。正常更新，用于更新逻辑。细心的读者应该可以发现每创建一个JavaScript脚本时，脚本中会默认添加这个方法。此方法每帧都会由系统自动调用一次。
- `function LateUpdate() {}`。推迟更新，此方法在`Update()`方法执行完后调用，同样每一帧都调用。
- `function Awake(){}`。脚本唤醒，此方法为系统执行的第一个方法，用于脚本的初始化，在脚本的生命周期中只执行一次。
- `function FixedUpdate() {}`。固定更新，在Unity导航菜单栏中，点击 “Edit” → “Project Settings”→“Time”菜单项后，右侧的Inspector视图将弹出时间管理器，其中“Fixed Timestep”选项用于设置`FixedUpdate()`的更新频率，更新频率默认为0.02s，如图4-7所示。

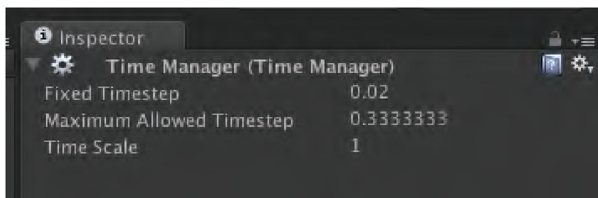


图4-7 设置时间界面

  固定更新常用于移动模型等操作。因为固定更新每一帧调用的时间相隔都是完全一样的，所以模型的移动过程会比较均匀。

- `function Start() {}`。此方法在`Awake()`方法之后、`Update()`方法之前执行，并且只执行一次。
- `function OnDestroy() {}`。当前脚本销毁时调用。
- `function OnGUI() {}`。绘制界面。这个方法大家应该不会陌生，因为在第3章中已经做过很多例子了。它和`Update()`方法一样，每一帧都在调用，只是它是用来绘制界面的。

4.3 利用脚本来操作游戏对象

在Unity场景中出现的所有实体都属于游戏对象，比如系统自带的立方体、球体以及由美工制作的.Fbx游戏模型等。游戏对象与脚本联系非常紧密，因为游戏对象之间的一切交互都需要使用脚本来完成。

下面我们开始学习如何使用脚本来操作游戏对象。使用脚本来调用游戏对象的方式有两种，它们是：可以将脚本绑定在一个游戏对象身上，也可以在代码中动态绑定脚本和删除脚本。任何一个游戏对象都可以同时绑定多条游戏脚本，并且这些脚本互不干涉，各自完成各自的生命周期。

4.3.1　创建游戏对象

创建游戏对象的方式共有以下两种。第一种为将模型预先放入Hierarchy视图中，然后在场景视图中任意拖动该模型在3D世界中的位置，运行游戏后该模型就会出现在Game视图中。第二种为在代码中根据条件动态创建与删除游戏对象，这种处理方式灵活性比较高。本节中我们将重点介绍第二种方式，如何使用代码来创建游戏对象。

如图4-8所示，本例在游戏视图中添加了两个按钮："创建立方体"按钮和"创建球体"按钮。点击其中一个按钮，将在游戏中动态添加立方体对象或球体对象。为了让创建的立方体对象与球体对象感应物理碰撞，我们需要为其添加刚体组件。本例的代码如代码清单4-1所示。

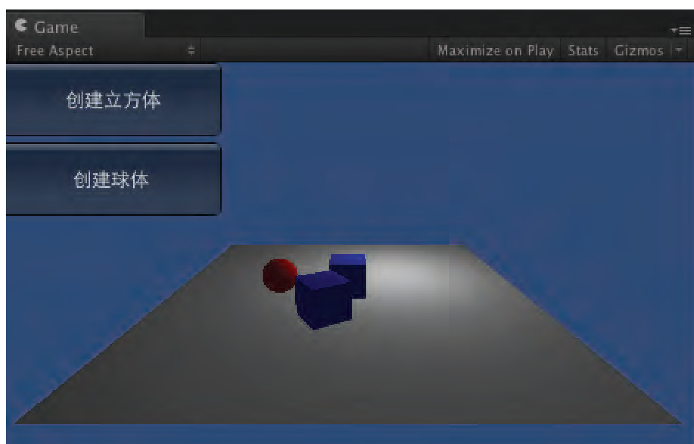


图4-8　创建游戏对象

代码清单4-1　Script_04_01.js文件

```
function OnGUI()
{

    if(GUILayout.Button("创建立方体",GUILayout.Height(50)))
    {
        //设置该模型默认为立方体
        var objCube = GameObject.CreatePrimitive(PrimitiveType.Cube);
        //给此对象添加一个刚体，用于物理感应
        objCube.AddComponent(Rigidbody);
        //设置这个游戏对象的名称
        objCube.name="Cube";
        //设置此模型材质的颜色
        objCube.renderer.material.color = Color.blue;
        //设置此模型的坐标
        objCube.transform.position = new Vector3(0.0f,10.0f,0.0f);
    }

    if(GUILayout.Button("创建球体",GUILayout.Height(50)))
```

```
    {
        //设置该模型默认为球体
        var objSphere = GameObject.CreatePrimitive(PrimitiveType.Sphere);
        //给此对象添加一个刚体，用于物理感应
        objSphere.AddComponent(Rigidbody);
        //设置这个游戏对象的名称
        objSphere.name="Sphere";
        //设置此模型材质的颜色
        objSphere.renderer.material.color = Color.red;
        //设置此模型的坐标
        objSphere.transform.position = new Vector3(0.0f,10.0f,0.0f);
    }

}
```

运行游戏后点击任意按钮，创建的立方体或球体将自动感应物理效果。因为创建的游戏对象悬浮在空中，所以对象创建完毕后它们将执行自由落体运动。下面简要介绍一下本例中几个重要的方法与引用。

- `GameObject.CreatePrimitive()`方法：用于创建一个原始游戏对象，其参数可设置为立方体、球体、圆柱体等系统默认提供的游戏对象。
- `AddComponent()`方法：用于给该游戏对象添加一个组件。
- `renderer.material.color`：设置渲染材质的颜色或者贴图。
- `transform.position`：设置该游戏对象的位置。

4.3.2 获取游戏对象

在脚本中获取游戏对象的方式一共有三种：第一种为通过对象名称获取，第二种为通过标签（tag）获取单个游戏对象，第三种为通过相同标签获取多组游戏对象。下面我们分别来学习如何使用这三种方式获取游戏对象。

1. 通过对象名称获取对象

如图4-9所示，在Hierarchy视图中添加立方体（Cube）与球体（Sphere）游戏对象，然后将球体对象拖曳至“Object”选项中，这表示Sphere为Object的子对象。

在代码中，使用`Find()`方法传入对象的完整路径名称即可在代码中获取游戏对象。本例中立方体的路径为“Cube”，而球体的路径为“Object/Sphere”。

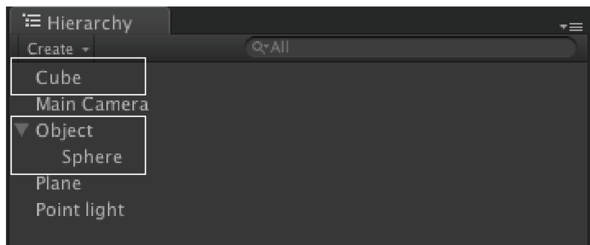


图4-9 获取游戏对象

本例中，我们通过游戏对象的完整路径获取当前游戏对象，通过游戏对象调用自身的旋转方法与销毁方法，具体代码如代码清单4-2所示。

代码清单4-2　Script_04_02.js文件

```
//立方体对象
private var objCube : GameObject;
//球体对象
private var objSphere : GameObject;
//是否旋转立方体
private var isCubeRoate  = false;
//是否旋转球体
private var isSphereRoate  = false;
//按钮提示信息
private var CubeInfo : String = "旋转立方体";
private var SphereInfo : String = "旋转球体";

function Start()
{
    //获取游戏对象
    objCube = GameObject.Find("Cube");
    objSphere = GameObject.Find("Object/Sphere");
}

function Update()
{
    //用户点击“旋转立方体”按钮时旋转模型
    if(isCubeRoate)
    {
        //当立方体对象不为null时旋转
        if(objCube)
        {
            objCube.transform.Rotate(0.0f,Time.deltaTime * 200,0.0f);
        }
    }

    //用户点击“旋转球体”按钮时旋转模型
    if(isSphereRoate)
    {
        //当球体对象不为null
        if(objSphere)
        {
            objSphere.transform.Rotate(0.0f,Time.deltaTime * 200,0.0f);
        }
    }

}

function OnGUI()
{

    //添加用于旋转立方体的按钮
    if(GUILayout.Button(CubeInfo,GUILayout.Height(50)))
    {
        if(!isCubeRoate)
        {
            isCubeRoate = true;
```

```
                CubeInfo = "停止旋转立方体";
            }else
            {
                isCubeRoate = false;
                CubeInfo = "旋转立方体";
            }
        }

        //添加用于旋转球体的按钮
        if(GUILayout.Button(SphereInfo,GUILayout.Height(50)))
        {
            if(!isSphereRoate)
            {
                isSphereRoate = true;
                SphereInfo = "停止旋转球体";
            }else
            {
                isSphereRoate = false;
                SphereInfo = "旋转球体";
            }
        }

        //添加用于销毁游戏对象的按钮
        if(GUILayout.Button("立即销毁模型",GUILayout.Height(50)))
        {
            //立即销毁立方体与球体对象
            Destroy(objCube);
            Destroy(objSphere);
        }
    }
```

在上述代码中，我们使用`GameObject.Find()`方法获得一个游戏对象，该方法的参数为游戏对象在Hierarchy视图中的完整路径，该方法的返回值就是需要获取的游戏对象。通过获取到的这个游戏对象，调用`transform.Rotate()`方法即可实现该游戏对象自身旋转。直接调用`Destroy()`方法即可销毁游戏对象，该方法的参数为需要销毁的游戏对象。运行程序后，效果如图4-10所示。

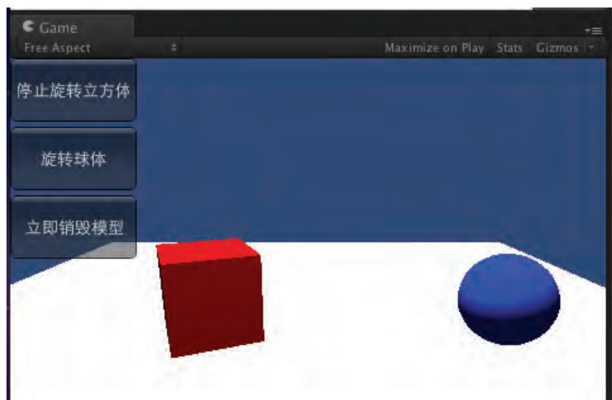


图4-10　游戏对象实例

2. 通过标签获取单个游戏对象

任何游戏对象都可以添加标签，这就好比给这个游戏对象起了一个小名一样。同样，通过标签也可以获得游戏对象。在获取游戏对象前，我们先学习如何给游戏对象添加一个标签。

在Hierarchy视图中点击创建的立方体，在右侧的Inspector视图中可以发现默认游戏对象的标签为“Untagged”（未标记）。点击标签后，弹出如图4-11所示的下拉列表，从中可以看到系统默认添加了7个标签，分别为“Untagged”、“Respawn”（重生）、“Finish”（完成）、“EditorOnly”（只编辑）、“MainCamera”（主摄像机）、“Player”（游戏者）和“GameController”（游戏控制器）。如果系统提供的标签未能满足项目的需求，还可点击“Add Tag...”选项（如图4-11所示），在打开的标签管理器中添加新标签。另外，也可通过Unity导航菜单栏中的“Edit”→“Project Settings”→“Tags”菜单项添加新标签。

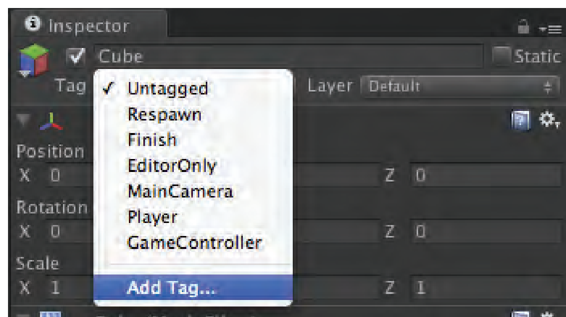


图4-11　添加游戏标签

选择“Add Tag...”菜单项后，程序将打开标签管理器界面。如图4-12所示，在标签管理器（Tag Manager）中“Element 0”选项右侧直接输入标签名称即可添加新标签，这里我们输入“MyTag”。此时在Inspector视图中立方体对应的标签下拉列表中已经出现MyTag这个标签选项（如图4-13所示），然后将该标签赋予立方体。需要注意的是，任何一个新标签添加完毕后，上方对应的“Size”与“Element”的长度都会自动累加。

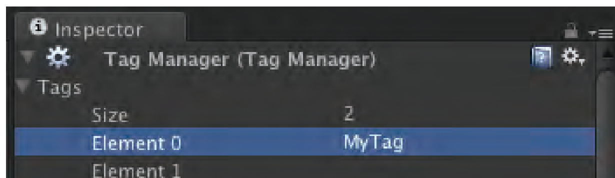


图4-12　标签管理器

下面我们在代码中使用`FindWithTag()`方法来获取标签对象，将标签名称作为参数传入后即可获得添加过“MyTag”标签的游戏对象：

```
//获取添加过“MyTag”标签的游戏对象
var obj : GameObject = GameObject.FindWithTag ("MyTag");
```

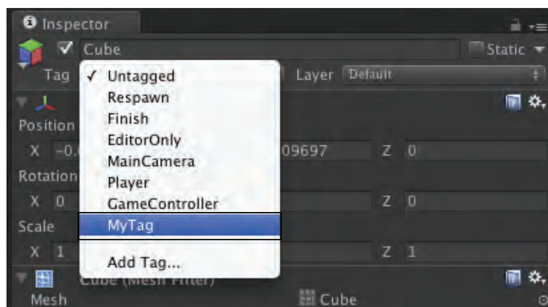


图4-13　添加的MyTag标签

注意　使用FindWithTag()方法只能获取一个游戏对象。如果程序中多个对象都添加了同样的标签，那么这个方法只能获取第一个添加过这个标签的对象。

3. 通过标签获取多个游戏对象

由于程序可能会存在很多相同的游戏对象，所以我们有必要将所有游戏对象的名称都记录下来。对多个不同的游戏对象，我们可添加一个相同的游戏标签。通过标签，可获取所有添加该标签的游戏对象。使用FindGameObjectsWithTag()方法并将相应的标签名称作为参数传入，即可返回一个游戏对象数组，其中包含具有这个标签的所有游戏对象。

下面我们举例说明通过标签获取多个游戏对象的方式。首先在Hierarchy视图中创建8个立方体对象，并在Inspector视图中给每一个立方体对象添加相同的标签“MyTag”，然后使用代码获取所有添加过“MyTag”标签的游戏对象。

运行游戏后，在代码中也可动态添加与修改标签，但是必须提前在标签管理器中注册该标签，否则在程序中修改标签时会抛出异常，提示无法找到该标签。本例中我们先获取所有游戏对象，然后动态修改其中的游戏标签，具体代码如代码清单4-3所示。

代码清单4-3　Script_04_03.js文件

```
function Start ()
{

    //得到包含MyTag标签的游戏对象数组
    var objs = GameObject.FindGameObjectsWithTag ("MyTag");
    //将5号元素的标签名称修改为TestTag
    objs[5].tag = "TestTag";

    //遍历所有游戏对象
    for (var obj in objs)
    {
        Debug.Log("以"+ obj.tag+"标签为游戏对象的名称 "+ obj.name);

        //判断标签的名称是否为TestTag
```

```
            if(obj.tag == "TestTag")
            {
                Debug.Log("这个标签为TestTag");
            }

            //判断该游戏对象是否包含TestTag这个标签
            if(obj.CompareTag("TestTag"))
            {
                Debug.Log("obj这个对象附带的标签为TestTag");
            }
        }

    }
```

运行程序后，效果如图4-14所示。

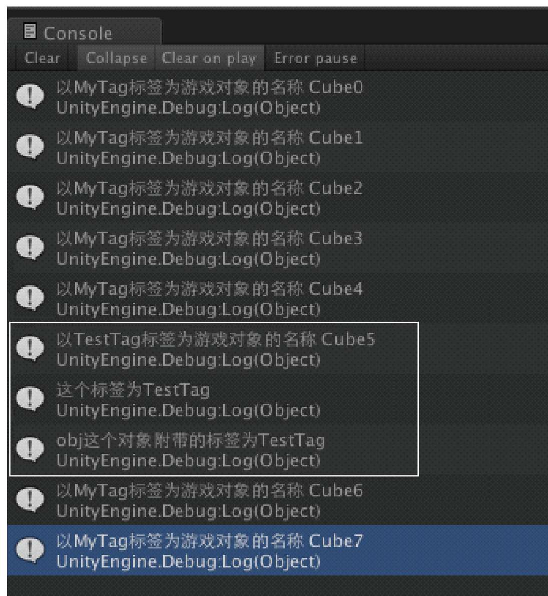


图4-14　游戏标签

4.3.3　添加组件与修改组件

新创建的游戏对象本身并不具备任何特性，为了让它具备一些功能，就必须给其添加游戏组件。游戏组件的种类非常多，常见的游戏组件有脚本类、网格类、粒子类、物理类、声音类和渲染类等。本节中我们将学习如何在代码中添加与修改游戏组件。添加游戏组件时，可以使用AddComponent()方法。由于组件自身没有对应的删除方法，需要使用父类执行Object.Destroy()方法才能删除它，其中该方法的参数为需要删除的游戏对象或游戏组件。在删除某一游戏对象时，将连带该对象中的所有组件一并删除。

本例中，我们首先在Scene视图中创建一个空的立方体对象，然后为其添加渲染组件。运行游戏后，在Game视图中点击“添加颜色”按钮或者“添加贴图”按钮将为该立方体对象添加颜色材质或者贴图材质，具体代码如代码清单4-4所示。

代码清单4-4　Script_04_04.js文件

```
//游戏对象
private var obj : GameObject;
//渲染器
private var render : Renderer;
//贴图
public var texture : Texture;

function Start()
{
    //获取游戏对象
    obj = GameObject.Find("Cube");
    //给当前对象添加一个脚本组件
    obj.AddComponent("Test");
    //获取该对象的渲染器
    render = obj.GetComponent("Renderer");
}

function OnGUI()
{
    if(GUILayout.Button("添加颜色",GUILayout.Width(100),GUILayout.Height(50)))
    {
        //修改渲染颜色为绿色
        render.material.color = Color.green;
        //为了避免残留，将贴图置空
        render.material.mainTexture = null;
    }
    if(GUILayout.Button("添加贴图",GUILayout.Width(100),GUILayout.Height(50)))
    {
        //为了避免残留，将贴图置空
        render.material=null;
        //添加组件贴图
        render.material.mainTexture = texture;
    }

}

function Update()
{
    //更新逻辑
}
```

在上述代码中，`render.material`引用为当前脚本绑定对象的材质，直接为其赋值即可修改对象材质。`render.material.color`引用为材质的颜色，`render.material.mainTexture`引用为材质的贴图。运行上面的代码后，效果如图4-15所示。

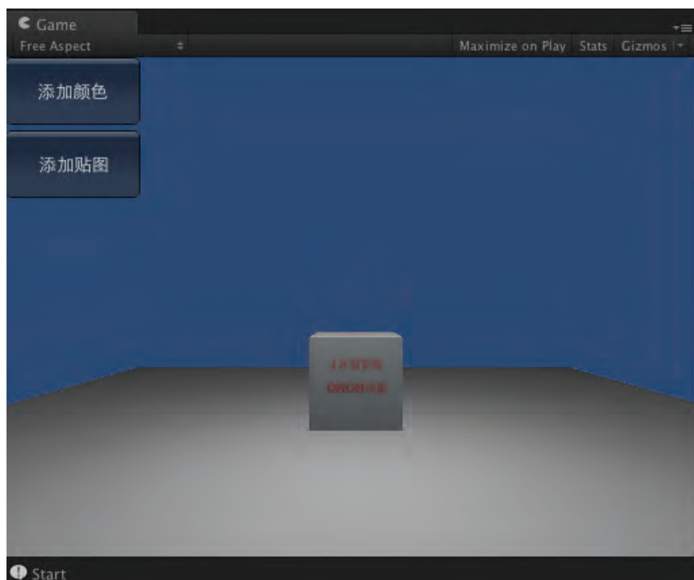


图4-15　添加组件

4.3.4　发送广播与消息

游戏对象是可以相互继承与派生的。依照这个原理，从父对象到子对象之间的继承数量就是任意的。在游戏对象之间使用广播传递消息非常普遍。我们知道，一个游戏对象可以绑定多条脚本，那么实现这些脚本之间的交互就可以使用广播机制。本节中，我们将学习游戏对象之间的广播发送与接收。

首先在场景中创建父游戏对象“GameObject”，它分别派生子对象“CubeA0”与“CubeA1”，其中“CubeA0”的子对象为“CubeB0”，“CubeB0”的子对象为“CubeC0”，而“CubeA1”的子对象为“CubeB1”，“CubeB1”的子对象为 “CubeC1”，如图4-16所示。可以看到，它们之间具有相互继承的关系。本例中，我们将向这6个子游戏对象绑定各自的游戏脚本，用来接收与发送消息。

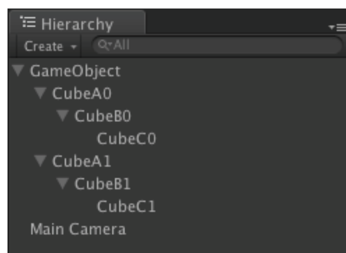


图4-16　继承关系

游戏对象之间发送的广播与消息大致可分为3种：第一种为向子对象发送，将发送至该对象的同辈对象或者子孙对象中；第二种为给自己发送，发送至自身对象；第三种为向父对象发送，发送至该对象的同辈或者父辈对象中。本例代码如代码清单4-5所示。

代码清单4-5　Script_04_05.js文件

```
//向子类发送消息
gameObject.BroadcastMessage ("ReceiveBroadcastMessage", "A0-----BroadcastMessage()");
//给自己发送消息
gameObject.SendMessage ("ReceiveSendMessage", "A0-----SendMessage()");
//向父类发送消息
gameObject.SendMessageUpwards ("ReceiveSendMessage", "A0-----SendMessageUpwards()");

//接收父类发送的消息
function ReceiveBroadcastMessage(str : String)
{
    Debug.Log("A0----Receive" +str);
}
//接收自己发送的消息
function ReceiveSendMessage(str : String)
{
    Debug.Log("A0----Receive" +str);
}
//接收子类发送的消息
function ReceiveSendMessageUpwards (str : String)
{
    Debug.Log("A0----Receive"  +str);
}
```

从上面的代码中可以看出，发送消息的方法都有两个参数：第一个参数为接受消息的方法名称，第二个参数为一个`object`类型的对象，可将任意类型作为参数附加在消息中。接收消息的方法具有一个参数，为`object`对象。

在实际开发中使用最多的就是`SendMessage()`方法。通过游戏对象可直接调用该方法，返回消息后，该对象身上绑定的所有脚本都可以接收到这个消息。

4.3.5　克隆游戏对象

克隆游戏对象与创建游戏对象在效果的呈现方式是完全一样的，但是从执行效率上来讲，克隆游戏对象的效率要高。使用脚本克隆游戏对象在游戏中应用非常广泛，常用于一些完全相同并且数量庞大的游戏对象，比如游戏中发射的子弹对象，每一颗子弹对象是完全一样的，每一次发射子弹都会新克隆一个子弹对象，并且让克隆的子弹对象完成自己的生命周期。

在代码中，需要使用`Instantiate()`方法克隆游戏对象。如图4-17所示，点击“开始克隆实例”按钮后，碗中的小球开始克隆对象并且完成克隆后5秒会自动销毁，具体代码如代码清单4-6所示。

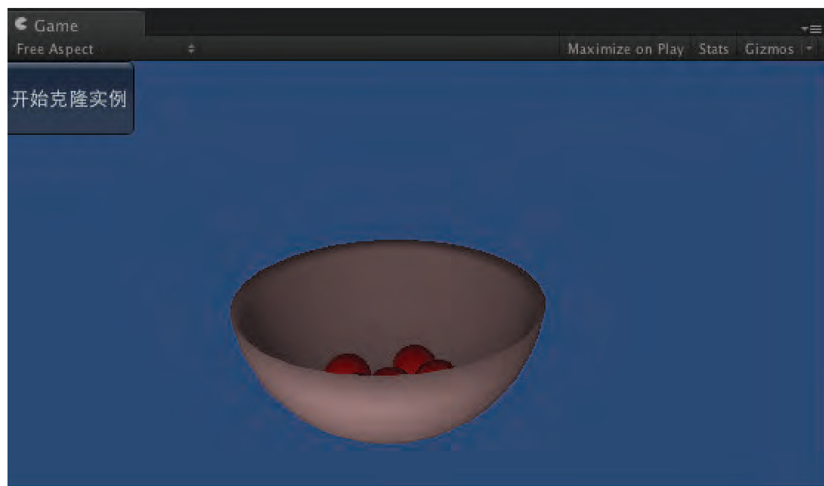


图4-17　克隆实例

代码清单4-6　Script_04_06.js文件

```
//球体对象
var obj : GameObject;

function Start()
{
    //获得球体对象
    obj = GameObject.Find("Sphere");
}

function OnGUI()
{

    if(GUILayout.Button("开始克隆实例",GUILayout.Height(50))){

        //克隆一个obj的实例
        var clone : GameObject = Instantiate(obj, obj.transform.position,
            obj.transform.rotation);
        //5秒后销毁该实例
        Destroy (clone, 5);
    }
}
```

Instantiate()方法的返回值就是克隆后的游戏对象，Destroy()方法的第一个参数为需要销毁的游戏对象，第二个参数表示延迟几秒后销毁。

4.3.6　脚本组件

在程序中，可动态地为某个游戏对象添加脚本组件。在程序中调用AddComponent()方法，

并将脚本的名称作为参数传入即可完成脚本组件的添加。如图4-18所示，本例中通过点击相关按钮实现脚本组件的添加与删除操作，具体代码如代码清单4-7所示。

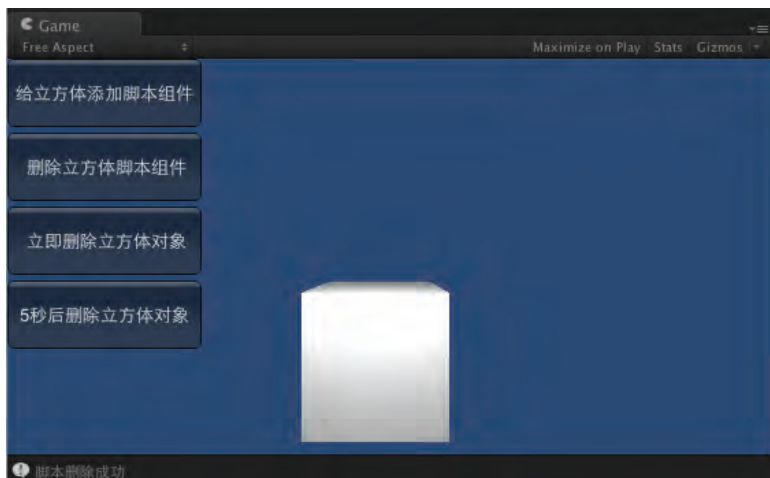


图4-18　脚本组件

代码清单4-7　Script_04_07.js文件

```
//立方体对象
var obj : GameObject;
function Start()
{
    //获得立方体对象
    obj = GameObject.Find("Cube");
}

function OnGUI()
{

    if(GUILayout.Button("给立方体添加脚本组件",GUILayout.Height(50)))
   {
        //添加cube_script脚本
        if(obj)
        obj.AddComponent("cube_script");
    }

    if(GUILayout.Button("删除立方体脚本组件",GUILayout.Height(50)))
   {
        //删除cube_script脚本
        if(obj)
        Destroy (obj.GetComponent ("cube_script"));
    }

    if(GUILayout.Button("立即删除立方体对象",GUILayout.Height(50)))
```

```
    {
        //删除立方体对象
        if(obj)
        Destroy (obj);
    }

    if(GUILayout.Button("5秒后删除立方体对象",GUILayout.Height(50)))
    {
        //5秒后删除立方体对象
        if(obj)
        Destroy (obj,5);
    }
}
```

测试添加脚本的代码（cube_script.js文件）如下所示:

```
function Start()
{
    Debug.Log("脚本添加成功");
}

function OnDestroy()
{
    Debug.Log("脚本删除成功");
}
```

本例中使用`obj.AddComponent("cube_script")`方法为游戏对象`obj`绑定一个名称为`cube_script`的脚本，然后新绑定的脚本将开始执行自己的生命周期。添加与销毁脚本时，将调用cube_script.js脚本文件中的`Start()`方法与`OnDestroy()`方法。

4.4　用脚本来控制对象的变换

在3D世界中，任何一个游戏对象在创建的时候都会附带Transform（变换）组件，并且该组件是无法删除的。

如图4-19所示，Transform面板中一共包含3个属性：Position（位置）、Rotation（旋转）和Scale（缩放）。在场景中使用移动工具来拖动和旋转模型，即可直接在Transform面板中看到编辑后的值，此外也可在Transform面板的编辑框中修改对象的位置、旋转方式和缩放的数值。本节中，我们将学习如何在代码中动态修改模型变换的数值。

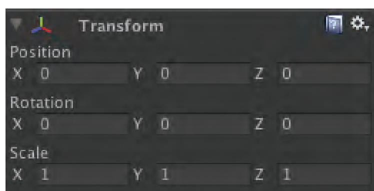


图4-19　Transform面板

4.4.1　改变游戏对象的位置

在3D世界中，任何一个模型的三维坐标都保存在Vector3容器中，该容器将记录模型在*x*轴、*y*轴和*z*轴方向的坐标。一旦在程序中修改该模型在Vector3容器中的坐标，那么Scene视图中模型的位置将发生改变。

在本示例中，我们将在游戏场景中创建一个立方体对象，并且使用GUI在界面中添加了3个拖动条，分别控制该模型在*x*轴、*y*轴和*z*轴上的坐标，拖动条滑块后即可看见立方体的位置发生了改变，效果如图4-20所示，具体代码如代码清单4-8所示。

代码清单4-8　Script_04_08.js文件

```
//立方体在x轴方向上的坐标
private var Value_X : float = 0.0f;
//立方体在y轴方向上的坐标
private var Value_Y : float = 0.0f;
//立方体在z轴方向上的坐标
private var Value_Z : float = 0.0f;
//立方体对象
private var obj : GameObject;

function Start()
{
    //得到立方体对象
    obj  = GameObject.Find("Cube");
}

function OnGUI()
{
    GUILayout.Box("移动立方体x轴");
    Value_X = GUILayout.HorizontalSlider(Value_X, -10.0f,
        10.0f,GUILayout.Width(200));
    GUILayout.Box("移动立方体y轴");
    Value_Y = GUILayout.HorizontalSlider(Value_Y, -10.0f,
        10.0f,GUILayout.Width(200));
    GUILayout.Box("移动立方体z轴");
    Value_Z = GUILayout.HorizontalSlider(Value_Z, -10.0f,
        10.0f,GUILayout.Width(200));

    //设置立方体的位置
    obj.transform.position = Vector3(Value_X,Value_Y,Value_Z);
    GUILayout.Label("立方体当前位置：" + obj.transform.position);

}
```

在上述代码中，我们使用`obj.transform.position`引用得到`obj`游戏对象在三维坐标系中的位置，它存储在Vector3容器中，该容器中保存着模型在*x*轴、*y*轴和*z*轴的坐标，所以说直接修改`position`容器的数值即可修改模型的位置。

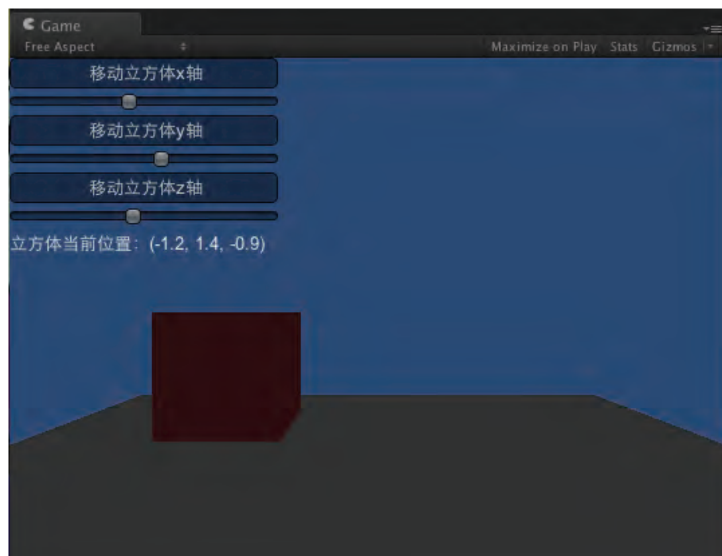


图4-20　模型的位置

4.4.2　旋转游戏对象

模型的旋转方式可分为两种：第一种为自身旋转，意思是模型沿着自己的*x*轴、*y*轴或*z*轴方向旋转；第二种为围绕旋转，意思是模型围绕着坐标系中的某一点或某一个游戏对象整体来做旋转。如图4-21所示，本例通过点击不同的按钮来设置模型的旋转模式，具体代码如代码清单4-9所示。

代码清单4-9　Script_04_09.js文件

```
//立方体对象
private var objCube : GameObject;
//圆柱体对象
private var objCylinder : GameObject;
//旋转速度
private var speed : int = 100;

function Start()
{
    //获得对象
    objCube = GameObject.Find("Cube");
    objCylinder = GameObject.Find("Cylinder");
}

function OnGUI()
{

    if(GUILayout.Button("立方体沿x轴旋转",GUILayout.Height(50)))
    {
```

```
        objCube.transform.Rotate(Vector3.right * Time.deltaTime * speed);
    }

    if(GUILayout.Button("立方体沿y轴旋转",GUILayout.Height(50)))
    {
        objCube.transform.Rotate(Vector3.up * Time.deltaTime * speed);
    }

    if(GUILayout.Button("立方体沿z轴旋转",GUILayout.Height(50)))
    {
        objCube.transform.Rotate(Vector3.forward * Time.deltaTime * speed);
    }

    if(GUILayout.Button("立方体围绕圆柱体旋转",GUILayout.Height(50)))
    {
        objCube.transform.RotateAround(objCylinder.transform.position,
            Vector3.up ,Time.deltaTime * speed);
    }
    GUILayout.Label("立方体旋转角度"+objCube.transform.rotation);

}
```


下面简要介绍一下本例中用到的一些重要方法和参数。

- ❑ transform.Rotate()：该方法用于设置模型绕自身旋转，其参数为旋转的速度与旋转的方向。
- ❑ transform.RotateAround()：该方法用于设置模型围绕某一个点旋转。
- ❑ Time.deltaTime：用于记录上一帧所消耗的时间，这里用作模型旋转的速度系数。
- ❑ Vector3.right：*x*轴方向。
- ❑ Vector3.up：*y*轴方向。
- ❑ Vector3.forward：*z*轴方向。

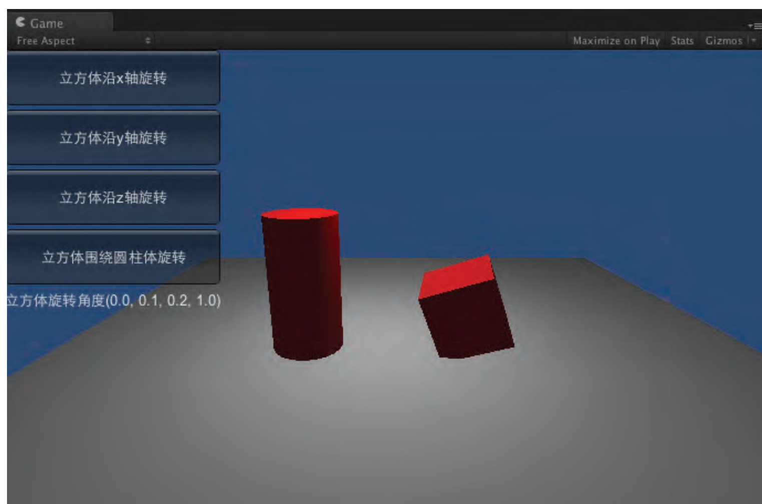


图4-21　模型的旋转

当模型绕自身旋转时，可以使用objCube.transform.Rotate()方法，其中objCube游戏对象为需要旋转的模型对象，该方法的参数为自身旋转的一个角度，角度包括x轴角度、y轴角度、z轴角度。当模型围绕某一点旋转时，则使用objCube.transform.RotateAround()方法，该方法包含两个参数，其中第一个参数表示所围绕的旋转点的位置，数据类型为Vector3，第二个参数为围绕旋转的角度，角度同样是x轴角度、y轴角度、z轴角度。

4.4.3　平移游戏对象

平移的含义是模型在原有位置的基础上继续移动，在代码中可以使用transform.Translate()方法来实现，此方法的唯一参数为平移模型的方向。

下面我们先在游戏场景中创建一个立方体对象，然后使用平移的方式点击“向前移动”、“向后移动”、“向左移动”和“向右移动”按钮来平移该立方体对象，效果如图4-22所示，具体代码如代码清单4-10所示。

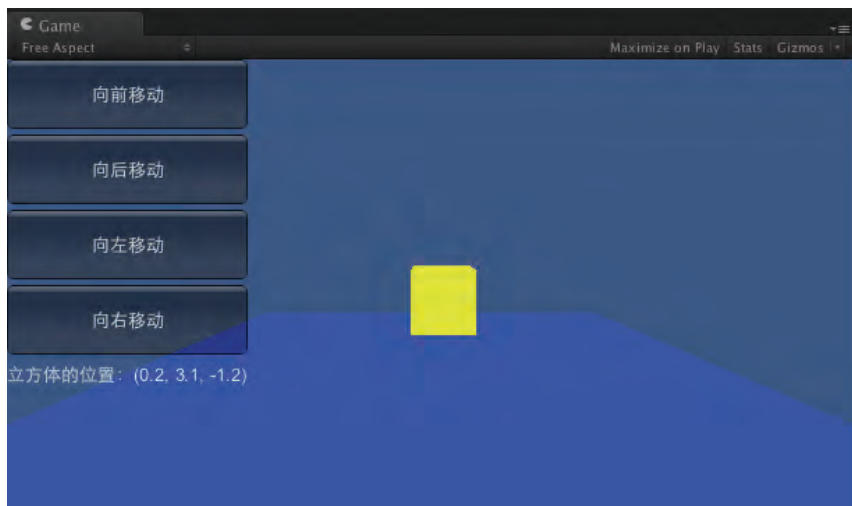


图4-22　平移模型

代码清单4-10　Script_04_10.js文件

```
//立方体对象
var obj : GameObject;

function Start()
{
    //获得立方体对象
    obj = GameObject.Find("Cube");
}
```

```
function Update()
{
}

function OnGUI()
{
    if(GUILayout.Button("向前移动",GUILayout.Height(50)))
    {

        obj.transform.Translate(Vector3.forward * Time.deltaTime);
    }

    if(GUILayout.Button("向后移动",GUILayout.Height(50)))
    {

        obj.transform.Translate(-Vector3.fwd * Time.deltaTime);
    }

    if(GUILayout.Button("向左移动",GUILayout.Height(50)))
    {

        obj.transform.Translate(Vector3.left * Time.deltaTime);
    }

    if(GUILayout.Button("向右移动",GUILayout.Height(50)))
    {

        obj.transform.Translate(Vector3.right * Time.deltaTime);
    }

    GUILayout.Label("立方体的位置："+obj.transform.position);
}
```

在上述代码中，我们使用`obj.transform.Translate()`方法来处理模型的平移，其中`obj`为需要平移的游戏对象，那么使用`obj`调用`transform.Translate()`方法的意思就是让“obj”对象开始平移。

4.4.4 缩放游戏对象

在Unity中,可以通过代码动态缩放游戏中的模型。主要有三种缩放方式：沿*x*轴缩放、沿*y*轴缩放、沿*z*轴缩放。每个轴都有自身的缩放系数，模型默认的缩放系数是1，就是模型原有大小，因此要在程序中实现模型的缩放，只需动态修改模型的缩放系数即可。

如图4-23所示，本例在游戏世界中创建一个立方体对象，通过在Game视图中拖动滑块来控制模型在三个轴上的缩放比例，本例代码如代码清单4-11所示。



图4-23　模型缩放前

拖动滑块后，立方体的比例发生改变，如图4-24所示。

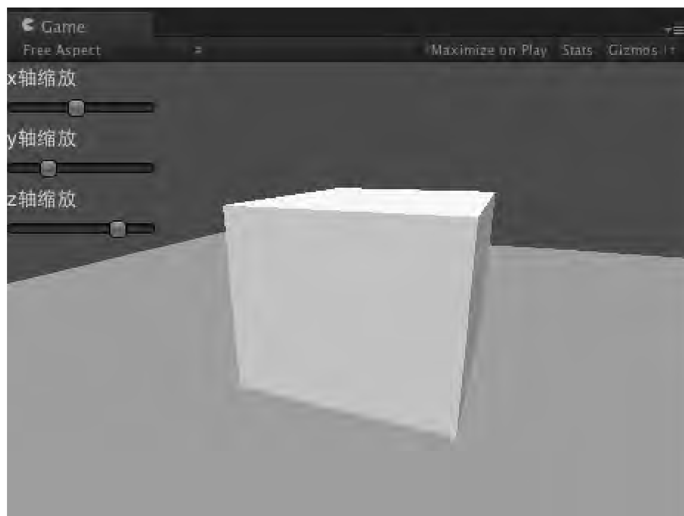


图4-24　模型缩放后

代码清单4-11　Script_04_11.js

```
var obj : GameObject;
//初始化缩放比例
var scaleX : float = 1.0;
var scaleY : float = 1.0;
```

```
var scaleZ : float = 1.0;
function Start ()
{
    //得到缩放模型对象
    obj = GameObject.Find("Cube");
}


function OnGUI ()
{
    GUILayout.Label("x轴缩放");
    scaleX = GUILayout.HorizontalSlider( scaleX, 1.0, 2.0,GUILayout.Width(100));
    GUILayout.Label("y轴缩放");
    scaleY = GUILayout.HorizontalSlider(scaleY, 1.0, 2.0,GUILayout.Width(100));
    GUILayout.Label("z轴缩放");
    scaleZ = GUILayout.HorizontalSlider( scaleZ, 1.0, 2.0,GUILayout.Width(100));
    //重新计算缩放比例
    obj.transform.localScale = Vector3(scaleX,scaleY,scaleZ);
}
```


在Game视图中拖动滑块后，得到*x*轴、*y*轴和*z*轴上模型的缩放系数，然后修改模型的`transform.localScale`值，即可重新设定模型的缩放比例。

4.5 用 C#编写脚本

目前，C#这门语言应用非常广泛，时下很流行的Windows Phone就是用这门语言开发的。Unity也支持使用C#语言去编写游戏脚本，不过目前不支持C＃的命名空间。从语法上看，C＃和JavaScript还是有一定区别的，并且部分代码的实现方式也不一样，但是C#语言的风格更符合Unity的编程思想。本节将带领读者去学习如何使用C#编写Unity游戏脚本。

4.5.1 继承MonoBehaviour类

任何一个游戏脚本（无论是JavaScript、C#还是Boo）都需要去继承`MonoBehaviour`这个类，只是在创建JavaScript脚本的时候，系统会将其类名与继承关系隐藏起来。

在本节中，我们先创建一个名为Test0.js的JavaScript脚本，然后立刻打开这个脚本，此时会发现系统自动帮我们生成`Update()`方法。在JavaScript脚本中，不存在`void`类型的方法，需要使用`function`作为关键字。JavaScript脚本生成的代码（Test0.js文件）如下所示：

```
function Update()
{
    //正常更新逻辑
}
```

为了区分JavaScript与C#，继续创建一个C#脚本，将其命名为Test1.cs。然后立刻打开它，可以发现，在C#脚本中系统已帮我们生成对应的继承关系：`Test1`继承`MonoBehaviour`类。

使用Test1脚本时，按照类名的命名规则，它必须与项目资源视图中该脚本的名称对应，否则

就无法成功绑定至游戏对象。与Java和C语言类似，在C#中返回任意类型需要使用void关键字。使用C#语言生成的代码（Test1.cs文件）如下所示：

```
using UnityEngine;
using System.Collections;

public class Test1 : MonoBehaviour
{
    void Start()
    {
        //开始方法
    }

    void Update()
    {
    //正常更新逻辑

    }
}
```

4.5.2　声明变量

C#与JavaScript语言在声明变量时是完全不一样的。使用JavaScript声明任何变量时，都 需要使用var关键字，并且需要在变量后面添加“:”以及具体对象类型与具体的数值。C#则使用对象类型加变量名的方式命名。

在下面的代码片段中，我们使用C#与JavaScript语言来实现相同的赋值操作，从中可清晰地看到两种语言声明变量的不同点。JavaScript语言的代码如下：

```
public var  i : int  = 0;
private var name : String[]  = ["test0","test1","test2"];
var obj : GameObject;
```

C#语言的代码如下：

```
public int i = 0;
public string []name= {"test0","test1","test2"};
public GameObject obj;
```

4.5.3　调用方法

在编写代码时调用方法很常见。因为不可能将所有代码都写在同一个方法中，所以使用方法将它们区别开来可以更好地管理代码。在下面的代码中，我们以一个经典的例子向读者诠释JavaScript与C#这两种语言在方法调用与方法返回上的区别。

下面先简要介绍JavaScript语言编写的代码，Test0.js 文件的代码如下：

```
//整型
var  i  : int;
//浮点型
```

```
var  f  : float;
//布尔型
var  b : boolean;
//字符串
var  str : String;

//设置整型
function  setInt(temp : int){
    i = temp;
}
//设置浮点型
function  setFloat(temp : float){
    f = temp;
}
//设置布尔型
function  setBoolean(temp : boolean) {
    b = temp;
}
//设置字符串
function  setString(temp : String) {
    str = temp;
}
//获取整型
function getInt() : int {
    return i;
}
//获取浮点型
function getFloat() : float {
    return f;
}
//获取布尔型
function getBoolean() : boolean {
    return b;
}
//获取字符串
function getString() : String {
    return str;
}
```

Main.js文件中的代码如下：

```
//立方体对象
var obj : GameObject;

function Start(){
    //获取立方体对象
    obj = GameObject.Find("Cube");
    //获取立方体绑定的脚本
    var script : Test0 = obj.GetComponent("Test0");

    //设置整型
    script.setInt(100);
    //设置浮点型
```

4

```
    script.setFloat(10.0f);
    //设置布尔型
    script.setBoolean(true);
    //设置字符串
    script.setString("Test");

    //获取信息并且打印
    Debug.Log(script.getInt());
    Debug.Log(script.getFloat());
    Debug.Log(script.getBoolean());
    Debug.Log(script.getString());
}
```

与之对应的C#语言代码如下所示，其中Test1.cs文件中的代码如下：

```
using UnityEngine;
using System.Collections;

public class Test1 : MonoBehaviour {
    //整型
    int  i;
    //浮点型
    float f;
    //布尔型
    bool b;
    //字符串
    string str;

    //设置整型
    public void setInt(int temp){
        i = temp;
    }
    //设置浮点型
    public  void setFloat(float temp){
        f = temp;
    }
    //设置布尔型
    public void  setBoolean(bool temp){
        b = temp;
    }
    //设置字符串
    public void  setString(string temp){
        str = temp;
    }
    //获取整型
    public int getInt() {
        return i;
    }
    //获取浮点型
    public float getFloat() {
        return f;
    }
    //获取布尔型
    public bool getBoolean() {
```

```
            return b;
        }
        //获取字符串
        public string getString() {
            return str;
        }

    }
```

Main.cs文件中的代码如下：

```
    using UnityEngine;
    using System.Collections;

    public class Main : MonoBehaviour {
        //立方体对象
        GameObject obj ;
        void Start() {

        //获取立方体对象
        obj = GameObject.Find("Cube");
        //获取Test1脚本
        Test1 script = obj.GetComponent<Test1>();

        //设置整型
        script.setInt(100);
        //设置浮点型
        script.setFloat(10.0f);
        //设置布尔型
        script.setBoolean(true);
        //设置字符串
        script.setString("Test");

        //获取信息并且打印
        Debug.Log(script.getInt());
        Debug.Log(script.getFloat());
        Debug.Log(script.getBoolean());
        Debug.Log(script.getString());

        }

    }
```

可以看到，JavaScript语言在调用方法时以`function`开头，然后提方法名和参数，而方法的返回类型在最后。C#语言在调用方法时以返回类型开头，接着是方法名与参数。

4.5.4 JavaScript与C#脚本之间的通信

虽然Unity官方说JavaScript与C#实现的脚本功能可以完全对等，但是根据开发习惯来说，项目中难免同时存在两种语言编写的脚本，此时就需要在它们两者之间进行通信。本例中，我们将学习这两种语言如何相互调用与传递参数，如图4-25所示。在本例中，点击“JavaScript调用C#”

或“C#调用JavaScript”按钮，可以实现两者之间的方法互调，其中JavaScript脚本调用C#脚本的代码如代码清单4-12所示。

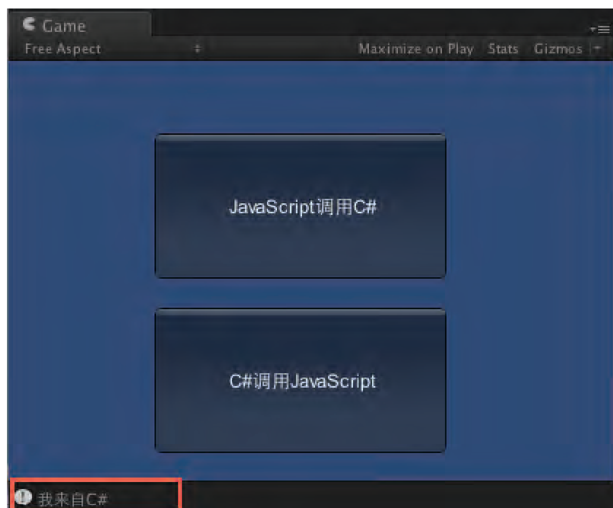


图4-25　相互调用

代码清单4-12　JS_test.js文件

```
function OnGUI()
{
    if(GUI.Button(Rect(100,50,200,100),"JavaScript调用C#"))
    {
        //获取C#脚本对象
        var cs = this.GetComponent("CS_test");
        //调用C#脚本中的方法
        cs.CallMe("我来自JavaScript");
    }

}

function CallMe(test : String)
{
    Debug.Log(test);
}
```

C#脚本调用JavaScript脚本的代码如代码清单4-13所示。

代码清单4-13　CS_test.cs文件

```
using UnityEngine;
using System.Collections;

public class CS_test : MonoBehaviour {
```

```
    void OnGUI()
    {

        if(GUI.Button(new Rect(100,170,200,100),"C#调用JavaScript"))
        {
            //获取JavaScript脚本对象
            JS_test jsScript = (JS_test)GetComponent("JS_test");
            //调用JavaScript脚本中的方法
            jsScript.CallMe("我来自C#");
        }

    }

    public void CallMe(string test)
    {
        Debug.Log(test);
    }
}
```


在脚本相互调用的时候，首先需要通过GetComponent()方法来获取脚本对象，然后通过脚本对象再去调用JavaScript或C#脚本中的方法。

如图4-26所示，本例将脚本JS_test.js与CS_test.cs同时绑定在摄像机中，这里需要注意的是JavaScript脚本必须放在“Standard Assets”文件夹中，否则无法编译通过。

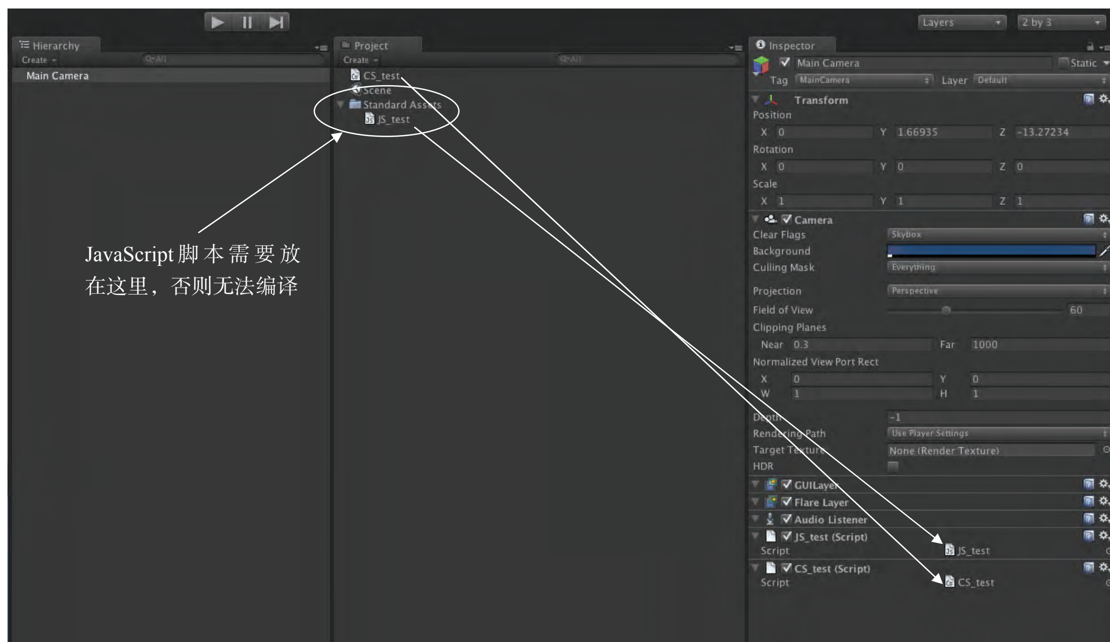


图4-26　脚本调用

4.6　工具类

Unity本身为开发者提供了很多方便开发的工具类，它们是由系统封装的一些功能与方法。为了避免开发者自己去手动实现，系统将它们封装得非常到位。这些工具类不仅功能强大，并且使用起来非常便捷，本节将带领读者了解Unity中一些常用的工具类。

学到这里，我相信读者已经正式入门Unity游戏开发了，后续章节我们将以C#语言进行讲解，因为C#语言更适合Unity的开发。

4.6.1　时间

Unity提供了`Time`类，这个类主要用来获取当前的系统时间。如图4-27所示，本例使用`Time`类将游戏运行中4个重要的时间数值打印了出来，具体代码如代码清单4-14所示。



图4-27　时间

代码清单4-14　Script_04_13.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_13 : MonoBehaviour
{

    void OnGUI()
    {
        GUILayout.Label("当前游戏时间：" + Time.time);
        GUILayout.Label("上一帧所消耗的时间：" + Time.deltaTime);
        GUILayout.Label("固定增量时间：" + Time.fixedTime);
        GUILayout.Label("上一帧所消耗固定时间：" + Time.fixedDeltaTime);
    }
}
```

下面简要说明这4个时间的具体含义。

- `Time.time`：从游戏开始后开始计时，表示截止目前共运行的游戏时间。
- `Time.deltaTime`：获取`Update()`方法中完成上一帧所消耗的时间。
- `Time.fixedTime`：`FixedUpdate()`方法中固定消耗的时间总和。`FixedUpdate()`每一

帧更新的时间可通过导航菜单栏“Edit”→“Project Settings”→“Time”菜单项去设置。

- Time.fixedDeltaTime：固定更新上一帧所消耗的时间。

4.6.2 等待

在程序中使用WaitForSeconds()方法可以以秒为单位让程序等待一段时间，此方法可直接使游戏主线程进入等待状态。该方法的返回值为IEnumerator类型，在需要等待的地方调用方法yield return new WaitForSeconds(2)，该方法中的参数表示主线程等待的秒数。

本例在Start()方法中让程序的主线程等待2秒后再继续执行，具体代码如代码清单4-15所示。

代码清单4-15 Script_04_14.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_14 : MonoBehaviour
{

    IEnumerator Start()
    {
        Debug.Log("开始等待：" + Time.time);
        yield return new WaitForSeconds(2);
        Debug.Log("结束等待：" + Time.time);
    }
}
```

如果要在某个方法中执行等待事件，那么在调用它的方法中同样要添加IEnumerator作为方法的返回类型，并且在调用的方法中执行return方法，代码如下所示:

```
using UnityEngine;
using System.Collections;

public class Script_13 : MonoBehaviour
{

    IEnumerator Start()
    {
        //等待
        return Test();
    }

    //返回等待时间
    IEnumerator Test()
    {
        Debug.Log("开始等待：" + Time.time);
        yield return new WaitForSeconds(2);
        Debug.Log("结束等待：" + Time.time);
    }

}
```

4

WaitForSeconds()方法用于通知游戏主线程等待。一定要将该方法的返回类型修改为IEnumerator，否则无法实现等待。

4.6.3 随机数

在开发中，有时需要获取程序中的随机数，这可以使用Random.Range()方法实现，其中该方法的第一个参数为随机数的起始位置，第二个参数为获取的随机数的结束位置。下面我们将通过Random.Range()方法分别获得一个整型和浮点型的随机数，具体代码如代码清单4-16所示。

代码清单4-16 Script_04_15.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_15 : MonoBehaviour
{

    void Start()
    {
        int a  = Random.Range(0,100);
        float  b = Random.Range(0.0f,10.0f);

        Debug.Log("获取一个0-100之间的整型随机数" + a);
        Debug.Log("获取一个0.0f-10.0f之间的浮点型随机数" + b);
    }


}
```

获取的随机数可以是整型，也可以是浮点型，使用方法int a = Random.Range(0,100)即可获取一个0到100之间的整型随机数。

4.6.4 数学

Unity帮开发者封装了一个数学类Mathf，使用它可以很轻松地帮我们解决复杂的数学公式。本节中，我们将向读者介绍Mathf类中一些常用的数学方法，具体如下所示。

- Mathf.Abs(int i)：返回一个绝对值，其返回值为整型或浮点型。
- Mathf.Clamp(int num,int min,int max)：返回一个限制值，限制参数1最小不超过参数2，最大不超过参数3。
- Mathf.Lerp(float start,float end, Time.time)：插入值，参数1表示开始的数值，参数2表示结束的数值，参数3表示所消耗的时间。
- Mathf.Sin(5)：返回正弦值。
- Mathf.Cos(5)：返回余弦值。
- Mathf.Tan(2)：返回正切值。

- Mathf.Max(0, 100)：返回两个数的最大值。
- Mathf.Min(0, 100)：返回两个数的最小值。
- Mathf.PI：圆周率。

要想了解其他方法，读者可自行查阅API阅读。

4.6.5　四元数

四元数是非常重要的工具类之一。在Unity中所有用到模型旋转的，其底层都是由四元数实现的，它可以精确地计算模型旋转的角度。下面我们将通过示例学习四元数的用法。如图4-28所示，在Game视图中点击“旋转固定角度”按钮，视图中的立方体对象将沿*y*轴直接旋转50°；点击“插值旋转固定角度”按钮，立方体将沿着*y*轴直接旋转50度，并且插入立方体旋转的时间，具体代码如代码清单4-17所示。

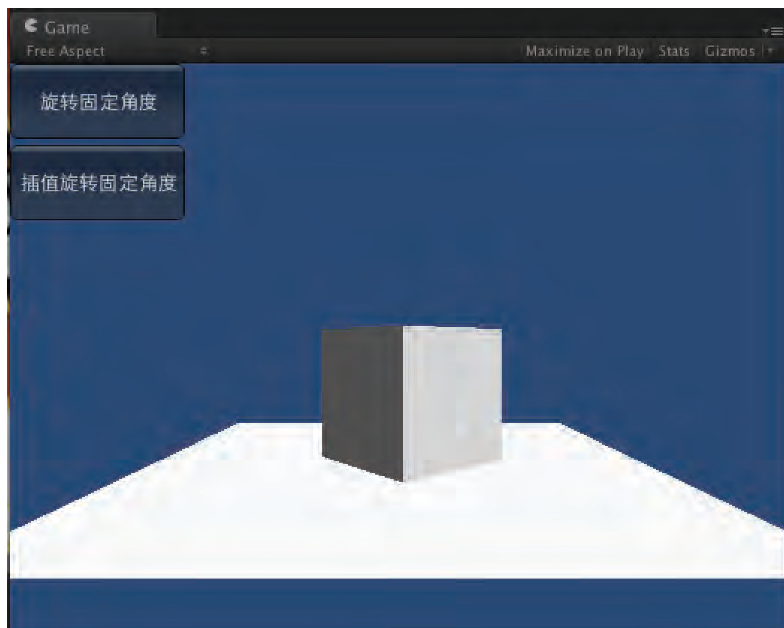


图4-28　四元数示例

代码清单4-17　Script_04_16.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_04_16 : MonoBehaviour
{
    //是否开始插值旋转
    bool isRotation = false;
```

```
    void OnGUI()
    {

        if(GUILayout.Button("旋转固定角度",GUILayout.Height(50)))
        {
            gameObject.transform.rotation = Quaternion.Euler(0.0f,50.0f,0.0f);
        }

        if(GUILayout.Button("插值旋转固定角度",GUILayout.Height(50)))
        {
            isRotation = true;
        }
    }

    void Update()
    {
        //开始插值旋转
        if(isRotation)
        {
            gameObject.transform.rotation = Quaternion.Slerp
                (gameObject.transform.rotation, Quaternion.Euler(0.0f,50.0f,0.0f),
                Time.time *0.1f);
        }
    }

}
```

在上述代码中，我们使用Quaternion.Euler()方法返回一个旋转的四元数，该方法中的参数表示旋转的三维角度。执行该方法后，将四元数赋值给立方体对象的旋转变量，即可在1帧内完成旋转。若想在一段时间内完成旋转，就需要使用Quaternion.Slerp()方法进行插值旋转，该方法的第一个参数表示模型旋转的起始角度，第二个参数表示模型旋转的结束角度，第三个参数表示旋转共消耗的时间。

4.7　游戏实例——小地图的制作

我相信大家都玩过MMORPG类的网游吧，主角在游戏世界中行走时，一般在屏幕右上角都会有一个区域来显示当前游戏场景的小地图。在小地图中，用一个比较小的点标记主角当前在地图中的位置，主角行走后该点也会发生改变。本示例中我们将讨论如何来制作这个游戏小地图。首先需要确认两个贴图，第一个是小地图的背景贴图，它应当是从*y*轴向下俯视截取出的贴图，第二个是主角位置贴图，它应当是在背景贴图之上的小型矩形。

如图4-29所示，本例使用面对象表示游戏地形，用立方体对象表示主角，使用GUI按钮来控制立方体对象的移动。右上角是游戏的小地图，中间的红色矩形表示当前主角在整个地形之上的位置。通过小地图的宽、高与真实地形的宽、高计算出缩放比例，最后根据这个缩放比例计算出小地图之上“主角”的位置。本例代码如代码清单4-18所示。

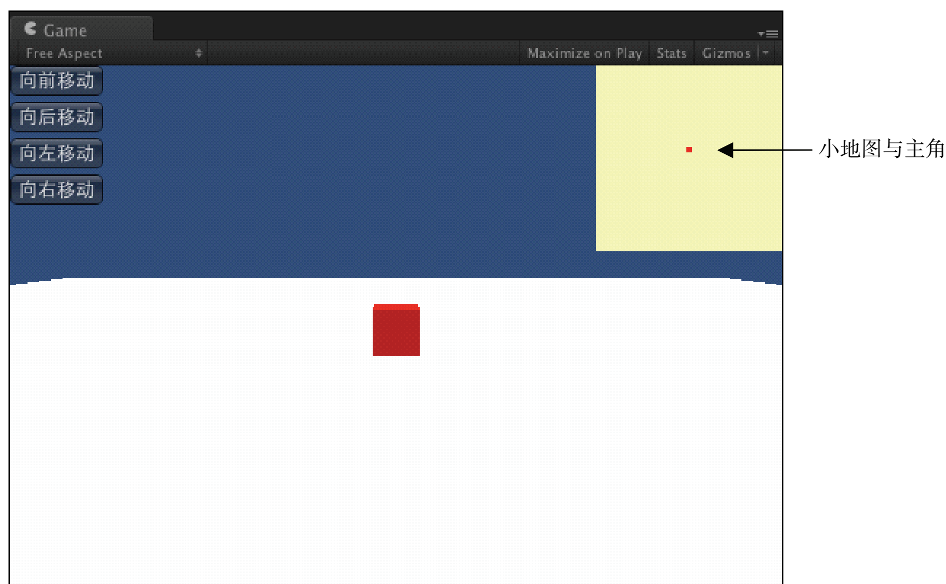


图4-29　游戏小地图

代码清单4-18　Script_04_17.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_04_17 : MonoBehaviour
{

    //大地图地形对象
    GameObject plane;
    //大地图主角对象
    GameObject cube;

    //大地图的宽度
    float mapWidth;
    //大地图的高度
    float mapHeight;
    //地图边界的检测数值
    float widthCheck;
    float heightCheck;

    //小地图主角的位置
    float mapcube_x =0;
    float mapcube_y = 0;


    //GUI按钮是否被按下
    bool keyUp;
```

```
bool keyDown;
bool keyLeft;
bool keyRight;

//小地图的背景贴图
public Texture map;
//小地图的主角贴图
public Texture map_cube;

void Start()
{
    //得到大地图对象
    plane = GameObject.Find("Plane");
    //得到大地图主角对象
    cube = GameObject.Find("Cube");
    //得到大地图默认宽度
    float size_x = plane.GetComponent<MeshFilter>().mesh.bounds.size.x;
    //得到大地图宽度的缩放比例
    float scal_x = plane.transform.localScale.x;
    //得到大地图默认高度
    float size_z = plane.GetComponent<MeshFilter>().mesh.bounds.size.z;
    //得到大地图高度的缩放比例
    float scal_z = plane.transform.localScale.z;

    //将原始宽度乘以缩放比例，计算出真实宽度
    mapWidth = size_x * scal_x;
    mapHeight = size_z * scal_z;

    //越界监测的宽度
    widthCheck = mapWidth / 2;
    heightCheck = mapHeight / 2;

    check();
}

void OnGUI()
{
    keyUp = GUILayout.RepeatButton("向前移动");

    keyDown = GUILayout.RepeatButton("向后移动");

    keyLeft = GUILayout.RepeatButton("向左移动");

    keyRight = GUILayout.RepeatButton("向右移动");

    //绘制小地图背景
    GUI.DrawTexture(new Rect(Screen.width - map.width,0,map.width,map.height),
        map);
    //绘制小地图上的“主角”
    GUI.DrawTexture(new Rect(mapcube_x,mapcube_y,map_cube.width,map_cube.
        height),map_cube);
}

void FixedUpdate()
```

```
{

    if(keyUp)
    {
        //向前移动
        cube.transform.Translate(Vector3.forward * Time.deltaTime *5);
        check();

    }

    if(keyDown)
    {
        //向后移动
        cube.transform.Translate(-Vector3.forward * Time.deltaTime *5);
        check();
    }

    if(keyLeft)
    {
        //向左移动
        cube.transform.Translate(-Vector3.right * Time.deltaTime *5);
        check();
    }

    if(keyRight)
    {
        //向右移动
        cube.transform.Translate(Vector3.right * Time.deltaTime *5);
        check();
    }
}

//越界检测
void check()
{
    //得到当前主角在地图中的坐标
    float x = cube.transform.position.x;
    float z = cube.transform.position.z;

    //当主角超过地图范围时，重新计算主角坐标
    if(x >= widthCheck)
    {
        x = widthCheck;
    }
    if(x <= -widthCheck)
    {
        x = -widthCheck;
    }
    if(z >= heightCheck)
    {
        z = heightCheck;
    }
    if(z <= -heightCheck)
    {
```

4

```
                z = -heightCheck;
            }

            cube.transform.position = new Vector3(x,cube.transform.position.y,z);

            //根据比例计算小地图中“主角”的坐标
            mapcube_x = (map.width/mapWidth * x) + ((map.width / 2) - (map_cube.width/2)) +
                (Screen.width - map.width);
            mapcube_y =map.height - ((map.height/mapHeight * z) + (map.height / 2));
        }
    }
```

由于我们还没有学习Unity游戏地形的制作，所以本例中使用“Plane”对象来代替地图。使用`mesh.bounds.size`得到当前对象默认的显示范围，接着通过游戏对象调用`transform.localScale()`方法得到对象的缩放比例，从而计算出地图的宽和高，最后根据小地图的宽和高计算出“主角”的坐标。

4.8　本章小结

本章首先介绍了Unity脚本编辑器MonoDevelop的用法，学习了如何使用MonoDevelop来调试程序。之后着重介绍了Unity脚本的生命周期。通过游戏脚本的学习，我们掌握了如何使用脚本来控制游戏对象。

接着详细介绍了使用JavaScript和C#编写游戏脚本的区别，其中JavaScript语言更适合初学者学习，而C#语言适合进阶阶段使用。最后学习了游戏开发中一些常用的工具类，它们都是由系统封装的类，非常好用。

第5章

游戏元素

在3D游戏世界中，通常会将很多丰富多彩的游戏元素融合至游戏中。游戏元素是制作游戏关卡的必备条件，它种类繁多并且作用也大不相同。游戏元素可分为常用元素与不常用元素两种，常用元素是游戏中一些比较重要的元素，它们需要使用脚本来实现一些特殊功能，比如玩家控制的主角对象、需要攻击的敌人、通关游戏的必要条件等，因此常用元素将直接影响游戏是否可以继续进行；而不常用元素在游戏世界中主要起装饰作用，比如游戏中的天空、云朵、树木和地形等，这些元素不会影响到游戏的主线，但是它们可以提升游戏的整体效果。

在游戏场景中，任何一款完美的游戏都需要使用这些不常用元素来配合，因为它们的存在往往是游戏画面的保证。在本章中，我们将学习Unity 3D中不常用的游戏元素，使用它们来制造属于我们自己的游戏世界。

5.1　游戏地形

玩过3D游戏的朋友应该对那些高低起伏的地形很有印象吧。无论是秀丽的山川还是辽阔的平原，地形元素都会很生动地出现在游戏世界中，这些高低起伏的地形是2D游戏无法媲美的。Unity中有一套非常好的地形编辑器，它可以让开发者实现游戏中任何复杂的地形，还可以制作地形上的一些元素，比如树木、草坪和石头等。

5.1.1　创建地形

下面开始学习如何创建游戏地形。首先打开Unity，在导航菜单栏中选择“Terrain”→“Create Terrain”菜单项（如图5-1所示），创建一个游戏地形。

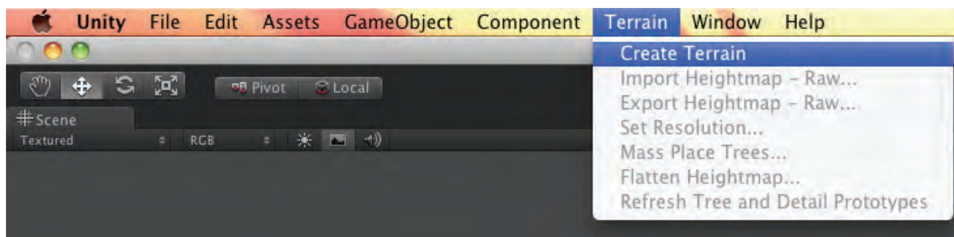


图5-1　“Terrain”菜单

此时游戏地形将以游戏对象的形式自动添加至游戏的Hierarchy视图中（如图5-2所示）。在Scene视图中，可观察到游戏地形是一个非常巨大的平面，之后的工作就是在这个巨大的平面中进行。在Hierarchy视图中，可以看到刚刚创建的地形为“Terrain”。依照苹果系统的操作习惯，使用鼠标选择文件名称后，按下回车键可以修改地形的名称。

在Hierarchy视图中选择游戏摄像机对象，即可在Scene视图的“Camera Preview”（游戏预览）中观察到当前的游戏地形。修改摄像机的位置与照射的方向，可以调节地形的显示区域。

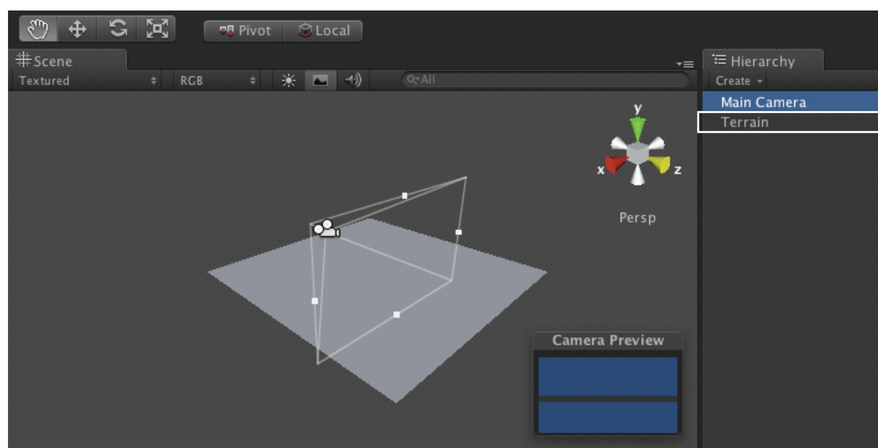


图5-2　默认地形

5.1.2　地形参数

地形参数包括地形的宽度、高度、长度、分辨率和高度图等。创建完地形后，可任意修改它们的参数。在Unity导航菜单栏中选择“Terrain”→“Set Resolution”菜单项，此时将弹出“Set Heightmap resolution”窗口（如图5-3所示），在每一个地形参数右侧直接输入数值即可修改它，然后在界面中点击右下角的“Set Resolution”按钮，即可将当前设置的所有地形参数应用到地形当中。

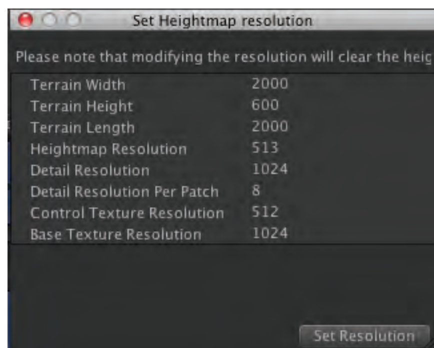


图5-3　设置地形参数

下面简要介绍一下在“Set Heightmap resolution”窗口中可编辑的地形参数。

- Terrain Width：地形总宽度。
- Terrain Height：地形总高度。
- Terrain Length：地形总长度。
- Heightmap Resolution：地形高度图的分辨率。
- Detail Resolution：细节分辨率，主要用于地形上的草或网格模型，其数值越高，显示效果就越好，同时也就需要更高的机器配置。这一个参数可根据硬件条件来设置。
- Detail Resolution Per Patch：每面片的细节分辨率。
- Control Texture Resolution：控制贴图的分辨率。
- Base Texture Resolution：相对贴图分辨率。

在实际开发中，较为常用的就是设置地形的宽度、高度以及长度。

5.1.3 编辑地形

到目前为止，我们创建的地形还是一个巨大的平面，下面我们将学习如何编辑地形，实现高低起伏的地形效果。首先在Hierarchy视图中选择“Terrain”地形，此时在右侧的Inspector视图中将显示用来编辑游戏地形的菜单，如图5-4所示。可以看到，地形菜单栏中一共含有7个按钮（第一个方框内），它们的含义分别为编辑地形高度、编辑地形特定高度、平滑过渡地形、地形贴图、添加树模型、添加草与网格模型、其他一些设置。

5



图5-4 编辑地形

在地形菜单栏中，光标默认指向第一个按钮编辑地形高度，这个按钮主要用来编辑地形中某区域的高度，此时的页面如图5-5所示，其中各个参数的含义如下所示。

- Brushes：地形的画笔。
- Brush Size：画笔宽度的取值范围。
- Opacity：画笔高度的取值范围。



图5-5　设置地形高度

选择画笔后，就可以在Scene视图中编辑地形的高度了。一般情况下，左右拖动鼠标即可编辑地形的宽度，上下拖动鼠标即可编辑地形的高度。

下面我们使用编辑地形工具在Scene视图中简单编辑一下当前的游戏地形，如图5-6所示。此时地形已经出现了凹陷与凸起的效果，不过目前的地形十分粗糙。因为默认的贴图是灰色的，效果非常难看。

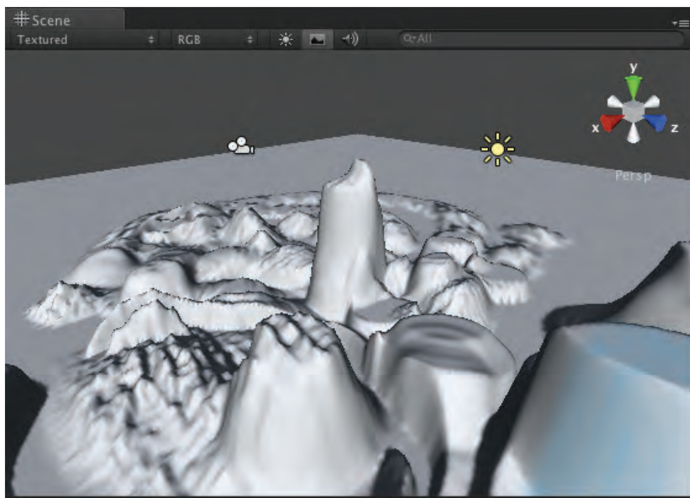


图5-6　当前地形效果

如图5-7所示，在地形菜单栏中点击第二个按钮（编辑地形特定高度），此时将打开设置地形特定高度页面，其中的设置项与编辑地形高度页面基本一样，只是多了一个“Height”（高度）设置选项，该参数用于设置地形的最大高度。

在编辑地形高度页面中，Opacity是最大高度，但是在编辑地形特定高度页面中，Height才是最大高度。比如，如果在编辑地形特定高度页面中，将“Opacity”属性设置为100，将“Height”属性设置为80，那么地形的最大高度就是80。

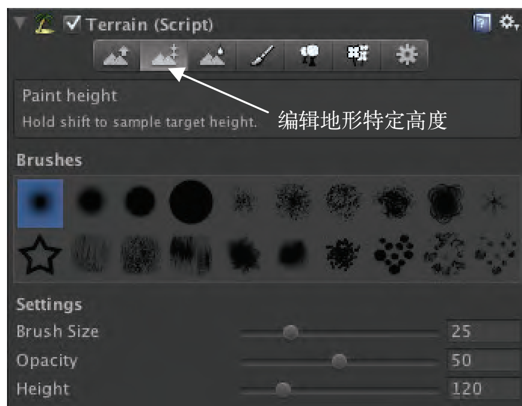


图5-7　编辑地形特定高度

如图5-8所示，在地形菜单栏中点击第三个按钮（平滑过渡地形），然后在下方画笔框中选择一个合适画笔，接着在Scene视图中拖动鼠标即可平滑过渡地形。平滑过渡编辑页面中各参数的含义如下所示。

- Brushes：平滑过渡画笔。
- Brush Size：画笔宽度的取值范围。
- Opacity：画笔高度的取值范围。

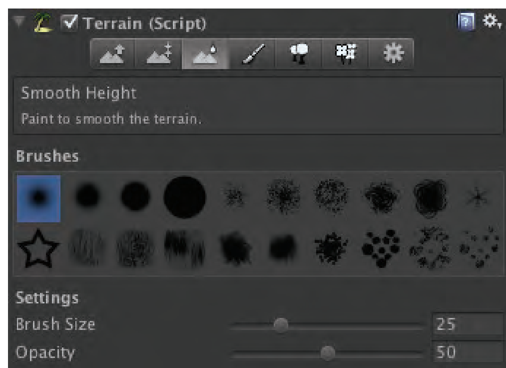


图5-8　平滑过渡编辑页面

设置平滑过渡后的地形如图5-9所示。与图5-6未设置平滑过渡的地形相比，两者呈鲜明的对比。

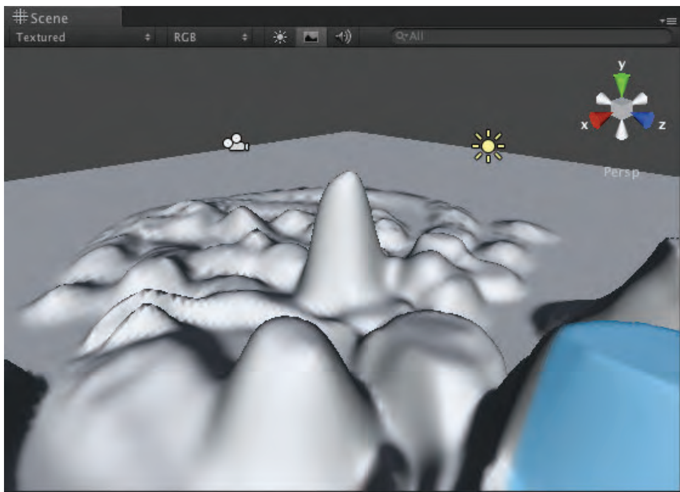


图5-9　设置平滑过渡

创建完地形后，系统会自动将“Terrain Collider”（地形碰撞）组件添加至地形当中，该组件可以让地形感应与其他物体之间的物理碰撞。给地形的表皮添加一个物理材质后，地形上的物体将根据地形物理材质中设置的碰撞参数而发生碰撞，比如摩擦力、弹力、动力等。下面先简要介绍一下地形碰撞组件中各参数的大致含义。

- Material：物理材质，设置模型与地形碰撞后的物理摩擦效果。
- Is Trigger：是否开启地形碰撞。
- Terrain Data：地形资源，连接Project视图中的地形文件。
- Create Tree Colliders：是否创建树木的碰撞。

5.1.4　地形贴图

在本节中，我们开始学习如何给地形添加贴图，让地形看起来更为美观。Unity提供了地形标准资源包，其中包含很多现成的地形资源以及贴图，它们都是免费供开发者使用的。本节我们就使用这个包中的地形资源来为地形添加漂亮贴图。

首先需要将地形资源包导入至当前工程中，具体方法为在Project视图中点击鼠标右键，选择“Import Package”→“Terrain Assets”菜单项，如图5-10所示。

此时地形的标准资源包将成功导入当前工程中。下面我们来学习如何给地形添加新的贴图。在地形菜单栏中点击第四个按钮（地形贴图），可以发现目前在“Textures”列表中没有任何地形贴图。下面我们先来学习如何将现有的图片资源添加到地形贴图中。在界面右下角点击“Edit Textures”（编辑贴图）按钮，此时将弹出一个选择列表，如图5-11所示，下面简要介绍各个菜单项的含义。

- ❑ Add Texture：添加地形贴图。
- ❑ Edit Texture：编辑地形贴图。
- ❑ Remove Texture：删除地形贴图。

图5-10　导入地形标准资源包

图5-11　弹出菜单

这里我们选择“Add Texture...”菜单项，此时程序将弹出“Add Terrain Texture”（添加地形贴图）界面，如图5-12所示，下面我们简要介绍一下该界面中各个参数的含义。

- ❑ Splat：在资源文件夹中选择一张地形贴图。
- ❑ Tile Size X：贴图x轴宽度。
- ❑ Tile Size Y：贴图y轴宽度。
- ❑ Tile Offset X：贴图x轴偏移量。
- ❑ Tile Offset Y：贴图y轴偏移量。

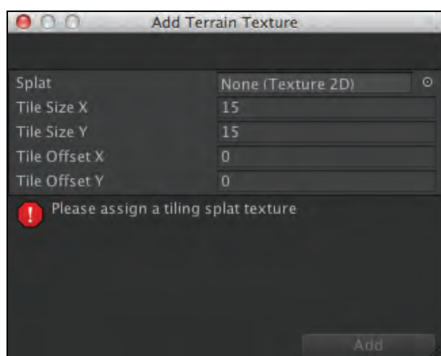


图5-12　添加地形贴图

单击Splat选项右侧的按钮，将弹出“Select Texture2D”页面，该页面中所有的贴图文件皆源于Project视图中，因为之前我们已将Unity提供的地形资源包已经导入至Project视图当中。在“Select Texture2D”页面中选择任意贴图资源，然后在左侧页面右下角处点击“Add”按钮即可将贴图添加至地形当中，如图5-13所示。

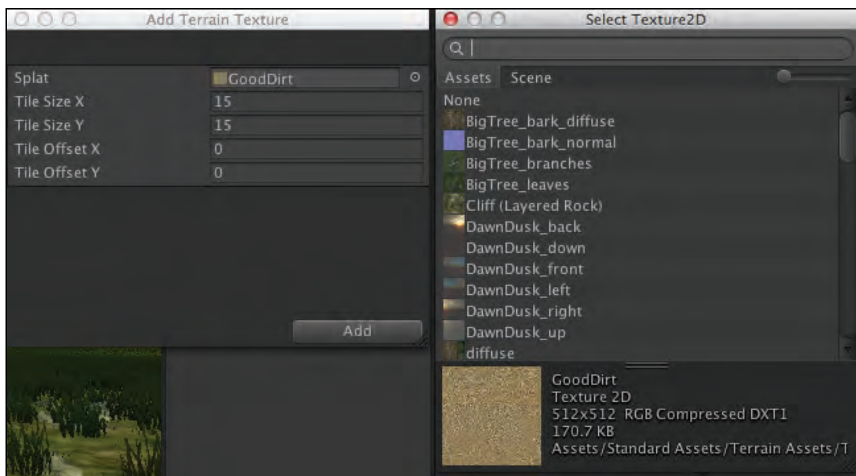


图5-13　选择资源界面

此时就可以渲染贴图了。首先选择一个渲染的贴图文件，然后选择一个渲染的画笔，最后在Scene视图中拖动鼠标即可在场景中渲染贴图，如图5-14所示。

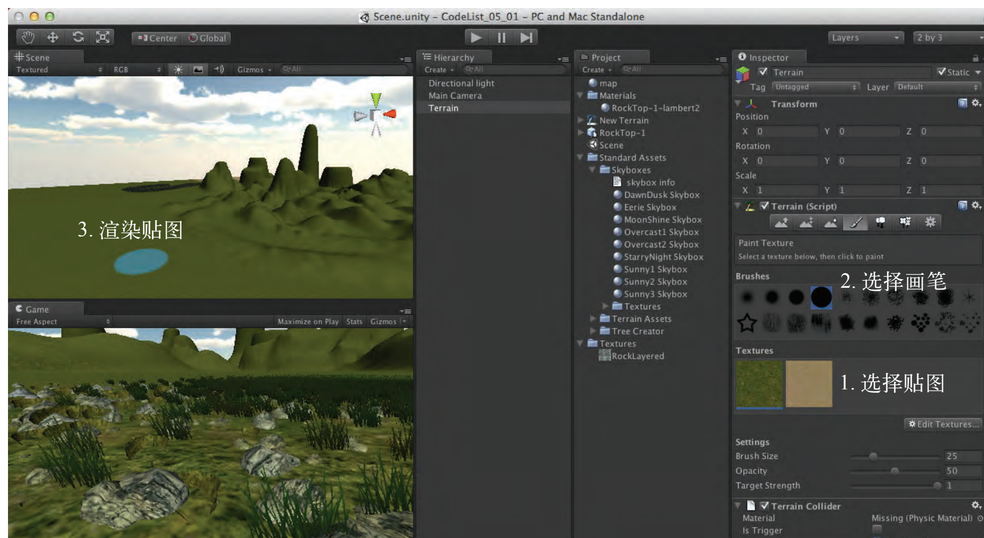


图5-14 渲染贴图

在Scene视图中经过一番涂鸦，地形中已经全部添加上了贴图，效果如图5-15所示。为了区分山丘与平地，本例共使用了两种地形贴图，山丘为绿色，平地为黄色。同一地形中可设定多个贴图资源，读者可根据自己的喜好自行设定。

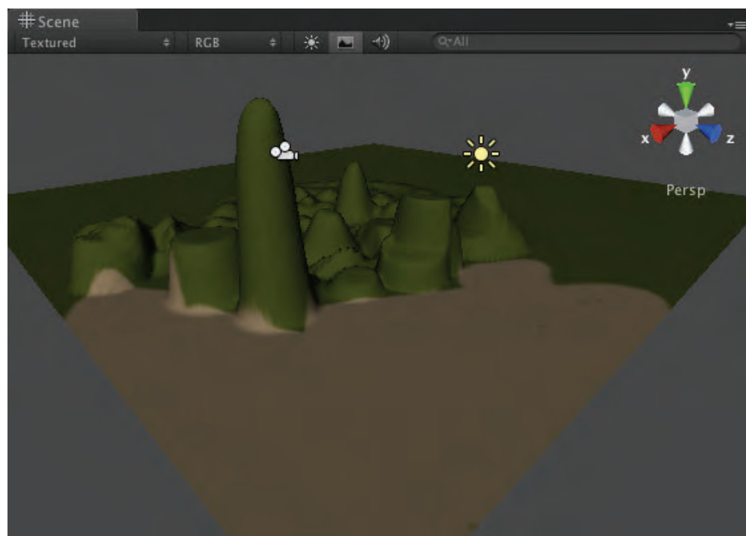


图5-15 地形贴图效果

5.2　地形元素

一般情况下，在游戏地形上会放置很多元素，这些元素与地形是分开的，主要包括树木、草地或自定义网格模型。地形元素在游戏世界中起装饰作用，所以不必将它们看作游戏对象，或者为它们添加复杂的游戏脚本。

5.2.1　树元素

在本节中，我们将学习如何将树元素添加至地形当中。首先导入系统提供的树木的标准资源包，具体操作方法是在Project视图中点击鼠标右键，然后从弹出的菜单中选择“Import Package”→“Tree Creator”菜单项，接着在地形菜单栏中点击第五个按钮（添加树模型），在打开的页面中批量添加树模型，如图5-16所示，其中“Settings”（设置）中各个参数的含义如下所示。

- ❑ Brush Size：树木的画笔大小，其值越大，绘制树木的范围就越大，反之则越少，取值范围为0到100。
- ❑ Tree Density：树之间的百分比，可理解为树与树之间的密度，取值范围为0到100。
- ❑ Color Variation：树之间颜色差的范围，取值范围为0到1。
- ❑ Tree Height：树的高度，其值越大表示树模型越高，取值范围为0到200。
- ❑ Variation：树与树之间的高度比例，取值范围为0到30。
- ❑ Tree Width：树的宽度，其值越大，树模型就越宽，反之则越窄，取值范围为0到200。
- ❑ Variation： 树与树之间的宽度比例，取值范围为0到30。

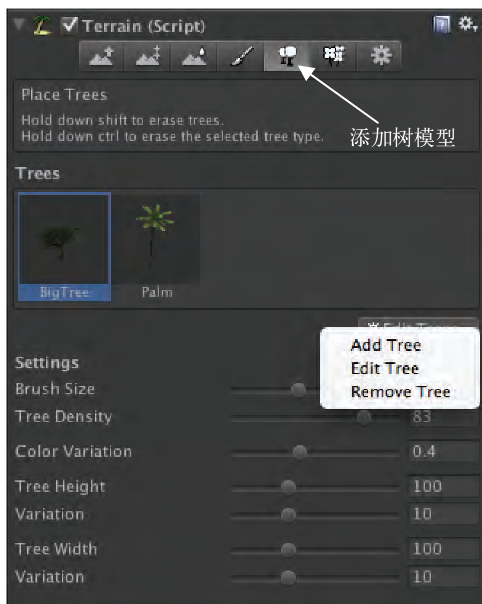


图5-16　添加树模型页面

点击右侧的“Edit Trees”（编辑树木）按钮，将弹出如图5-16所示的下拉列表，其中各个选项的含义如下所示。

❑ Add Tree：添加一个树模型。

❑ Edit Tree：编辑一个树模型。

❑ Remove Tree：删除树模型。

下面开始在工程中添加树模型。在如图5-15所示的下拉列表中选择“Add Tree”菜单项，程序将弹出选择模型页面，如图5-17所示。首先在“Select GameObject”页面中选择一个树的模型对象，然后在“Add Tree”页面中点击右下角的“Add”按钮完成添加。

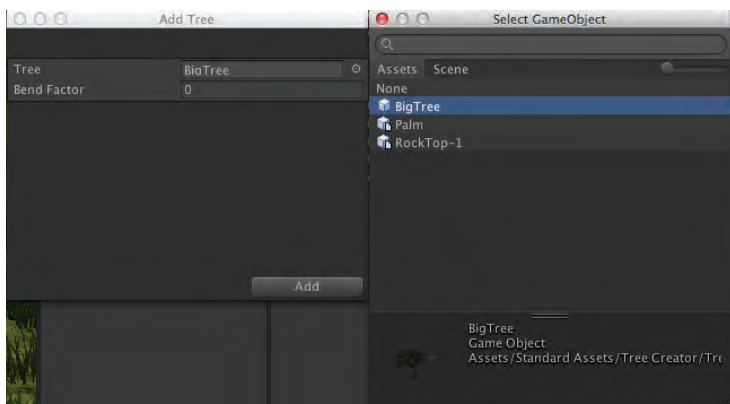


图5-17　选择模型

接着我们开始在场景中添加树。首先在右侧的Inspector视图中选择需要添加的模型，然后在Scene视图中使用画笔添加树模型即可，如图5-18所示。

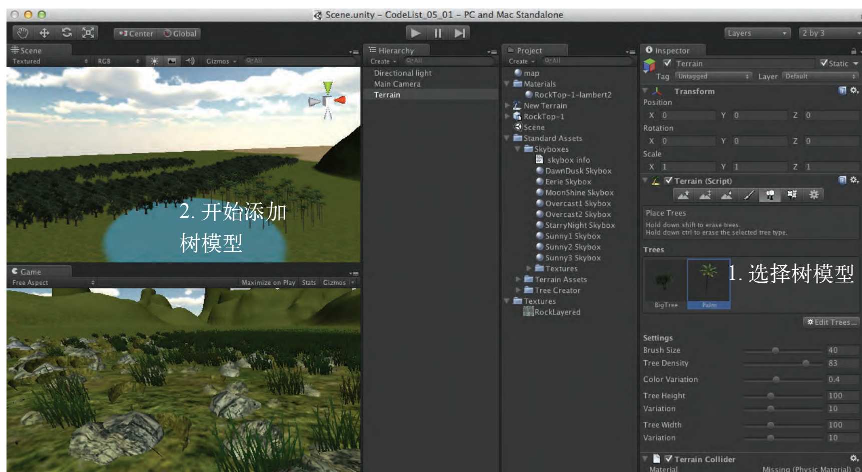


图5-18　添加树模型

5.2.2　草与网格元素

草与网格元素属于地形中的细节元素，其添加方法与树木的添加方法非常相似。如图5-19所示，首先在地形菜单栏中选择第六个按钮（添加草与网格模型），此时页面下方将显示一些细节信息，主要涉及画笔与贴图的一些属性，它们的含义如下所示。

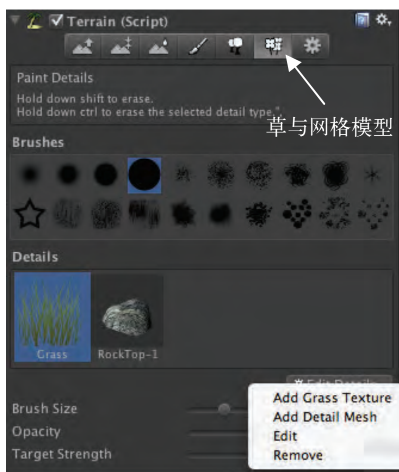


图5-19　添加草与网格模型

- ❑ Brush Size：画笔的大小。
- ❑ Opacity：绘制的高度。
- ❑ Target Strength：绘制的密度。

下面开始添加草或网格元素。点击页面右下角的“Edit Detail”（编辑细节）按钮，此时程序将弹出一个下拉列表，该下拉列表中各选项的含义如下所示。

- ❑ Add Grass Texture：添加草的贴图。
- ❑ Add Detail Mesh：添加自定义网格模型。
- ❑ Edit：编辑一个现有的模型。
- ❑ Remove：删除一个模型。

在弹出的列表中点击“Add Grass Texture”（添加草贴图）按钮，将打开如图5-20所示的“Add Grass Texture”页面，其中存在若干有关草贴图的选项，这些选项可用来编辑草的一些属性，下面简要介绍一下这些属性的含义。

- ❑ Detail Texture：选择草的贴图。
- ❑ Min Width：草的最小宽度（单位为米）。
- ❑ Max Width：草的最大宽度（单位为米）。
- ❑ Min Height：草的最小高度（单位为米）。
- ❑ Max Height：草的最大高度（单位为米）。

- ❑ Noise Spread：该数值越大，草干枯的范围就越大；数值越小，草干枯的范围就越小。
- ❑ Healthy Color：草正常的颜色。
- ❑ Dry Color：草干枯的颜色。
- ❑ Billboard：是否以主摄像机为中心旋转。

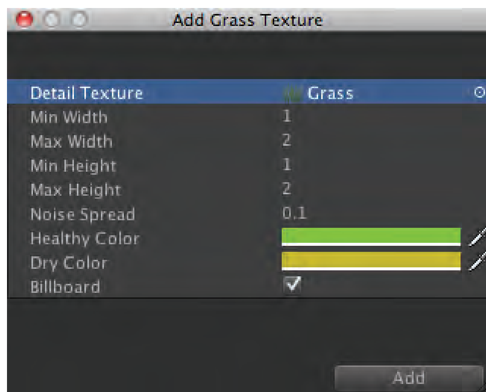


图5-20　“Add Grass Texture”页面

在弹出列表中点击“Add Detail Mesh”（添加网格）按钮，将打开用来添加自定义网格模型的“Add Detail Mesh”页面，如图5-21所示。本例中添加一个石头的模型作为自定义网格模型。同样，该页面含有若干选项，它们可用来编辑网格模型的一些属性，下面先简要介绍一下其中各个选项的含义。

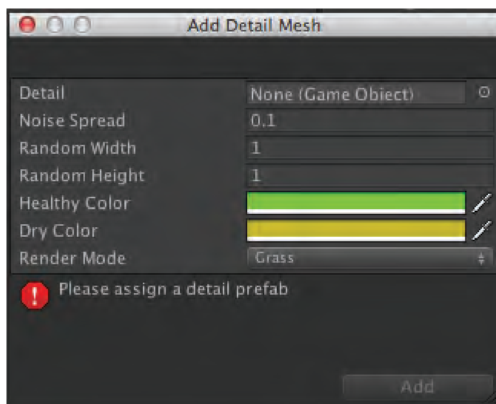


图5-21　添加网格模型页面

- ❑ Detail：网格模型的资源文件。
- ❑ Noise Spread：模型的范围大小。
- ❑ Random Width：随机宽度。

- ❑ Random Height：随机高度。
- ❑ Healthy Color：正常的颜色。
- ❑ Dry Color：模型枯萎的颜色。
- ❑ Render Mode：渲染模式。

根据提示，我们将草与网格元素加入当前工程中。如图5-22所示，首先在右侧的Inspector视图中选择需要绘制的模型，然后在Scene视图中即可开始添加草与石头的模型了。

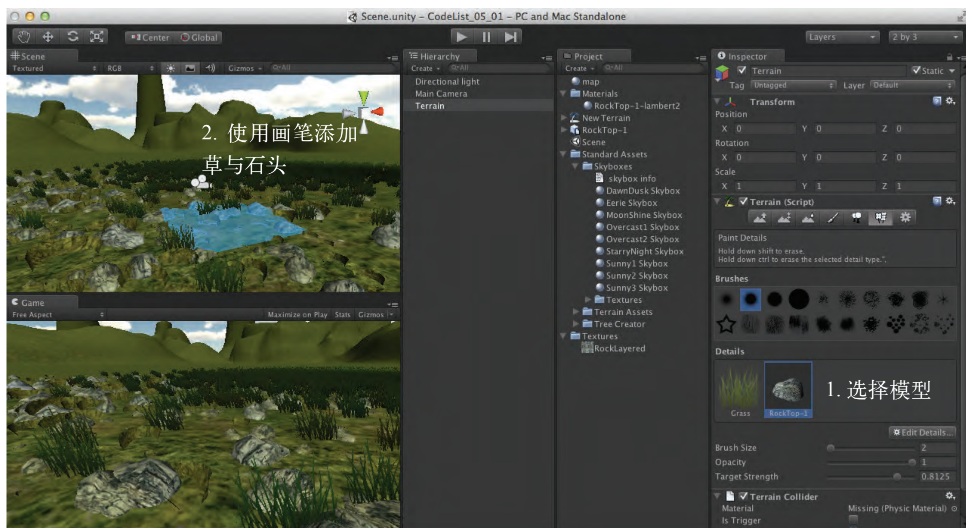


图5-22　添加草与石头的模型

5.2.3　其他设置

如图5-23所示，在地形菜单栏中点击第七个按钮（设置菜单项），在打开的页面中可设置一些辅助地形的选项，比如地形的容差、草与网格对象的显示、风对游戏场景的影响等。通过修改其中的参数，可以更精细地设置游戏地形。修改其中的选项后，直接运行游戏即可看到效果。下面我们先简要介绍一下这些参数的含义。

与Base Terrain（地形相关）相关的参数如下所示。

- ❑ Pixel Error：地形显示容差，其数值越小，棱角越清晰。
- ❑ Base Map Dist：地形贴图显示的精细度，数值越小表示越精细。
- ❑ Cast shadows：是否显示地形的阴影。

与Tree & Detail Objects（树与一些细节对象）相关的参数如下所示。

- ❑ Draw：是否显示草与网格模型。
- ❑ Detail Distance：根据摄像机的位置来设置细节模型显示的范围，数值越大表示同时显示的内容越多。

- Detail Density：设置细节模型显示的密度。
- Tree Distance：根据摄像机的位置来设置树显示的范围，数值越大表示同时显示的内容越多。
- Billboard Start：优化渲染效率，摄像机距离该模型一段距离时，不绘制整个网格而直接绘制一个贴图。
- Fade Length：树与网格显示的长度。
- Max Mesh Trees：网格显示的最大数量。

与Wind Settings（风的设置）相关的参数如下所示。

- Speed：风的速度。
- Size：风的长度及影响的范围。
- Bending：受风影响的草的数量。
- Grass Tint：草的颜色。



图5-23　其他设置

5.3　光源

在3D游戏中，光源是一个非常具有特色的游戏组件，为什么这么说呢？因为它可以提升游戏的画面质感。在新创建的场景中，默认是没有光源的，场景非常昏暗，所以开发中必须在场景中添加光源组件。

Unity引擎一共为开发者提供了3种不同的光源类型——点光源、聚光灯和平行光，它们可以模拟自然界中任何一种光。光源属于游戏对象，可在Scene视图中编辑它的位置以及光照的相关

参数。此外，光源还支持移动、旋转和缩放等操作。在实际开发中，大家可根据不同的场景而使用不同的光源。

5.3.1 点光源（Point Light）

顾名思义，点光源是在3D世界中从某一个点向周围扩散发出光的光源。如图5-24所示，点光源好像包围在一个类似球形的物体中，读者可将球形理解为点光源的照射范围，就像家里的灯泡可以照亮整个屋子一样。创建点光源的方式为在Hierarchy视图中点击“Create”→“Point Light”菜单项。

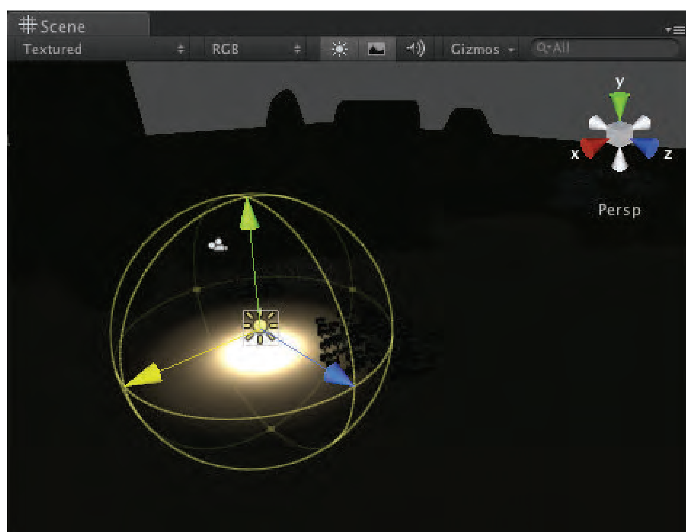


图5-24 点光源

创建完点光源后，在Hierarchy视图中选择该点光源对象，此时右侧的Inspector视图中将看到这个点光源的所有参数信息，如图5-25所示，下面先简要介绍一下这些参数的含义。

- Type：光源的类型。点击后面的小三角，可以发现共有3个选项：Point（点光源）、Directional（平行光）和Spot（聚光灯）。选择其中一项后，可切换到当前光的类型。
- Range：光照的影响范围。
- Color：光照的颜色。
- Intensity：光照的强度，可在原有光照的影响范围上缩小光照的面积。
- Cookie：设置贴图的阿尔法透明通道。将点光源看作立方体，可设置其六个面有不同的亮度，所以这里必须使用Cubemap贴图，其他2D贴图均无效。
- Shadow Type：光源投射的阴影类型。
- Draw Halo：是否在点光源中使用白雾效果。
- Flare：设置光源粒子效果。

- ❑ Render Mode：光源的渲染模式。
- ❑ Culling Mask：通过层可设置某些地图层不受光照影响。
- ❑ Lightmapping：设置光照贴图模式。

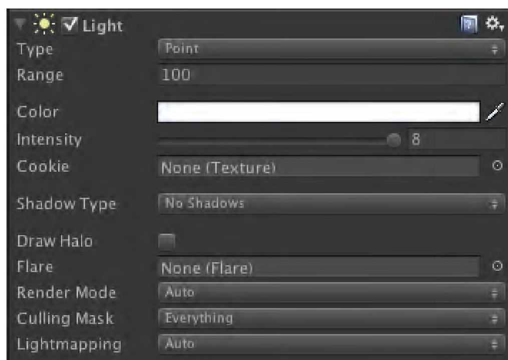


图5-25　点光源设置

5.3.2　聚光灯

聚光灯的原理很简单，它在3D世界中以某一个点为起点向以另一个点为圆心的平面发射一组平行光，以射线的形式照射在平面中，与手电筒的原理如出一辙（如图5-26所示）。

聚光灯在游戏中应用非常广泛，比如在第一人称游戏中，可将聚光灯绑定在主角身上，当玩家控制主角移动时，该光源也会跟着移动，始终照亮主角前方的路。创建聚光灯的方法如下：在Hierarchy视图中选择“Create”→“Spotlight”菜单项。

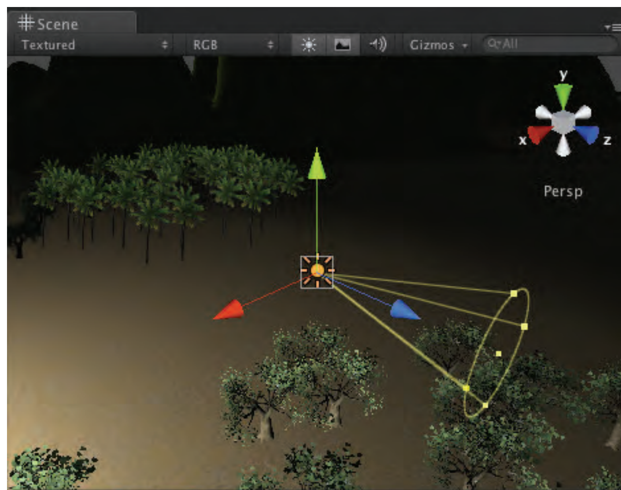


图5-26　聚光灯

创建完聚光灯后，在Hierarchy视图中选择该对象，此时在右侧的Inspector视图中可看到该聚光灯的相关参数，如图5-27所示，其中大多数设置项与前面介绍的点光源类似，唯一不同的是聚光灯中有“Spot Angle”（光照角度）选项，该数值可调节射线的光照范围。

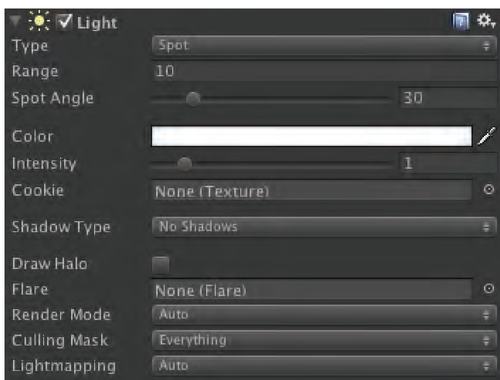


图5-27　聚光灯设置

5.3.3　平行光

平行光（Directional Light）的照射范围非常大，它可以照亮整个游戏世界，就好比自然界的太阳一样。在游戏开发中，室外场景必须设置平行光，否则游戏世界整体会非常黑暗。

使用它的时候，需要旋转其照射世界的角度。如图5-28所示，目前该光源的照射方向未旋转至地面上，所以在Scene视图中观察时，地面一片漆黑。使用旋转工具旋转该光源后，整个地面都亮了起来，如图5-29所示。

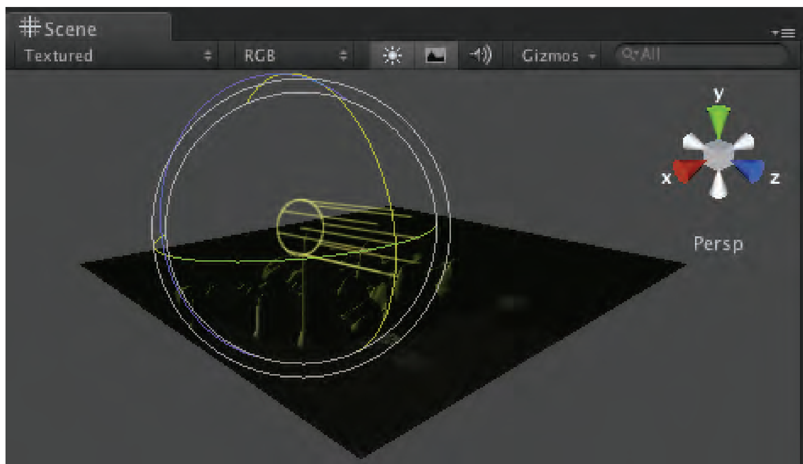


图5-28　未旋转的光源效果

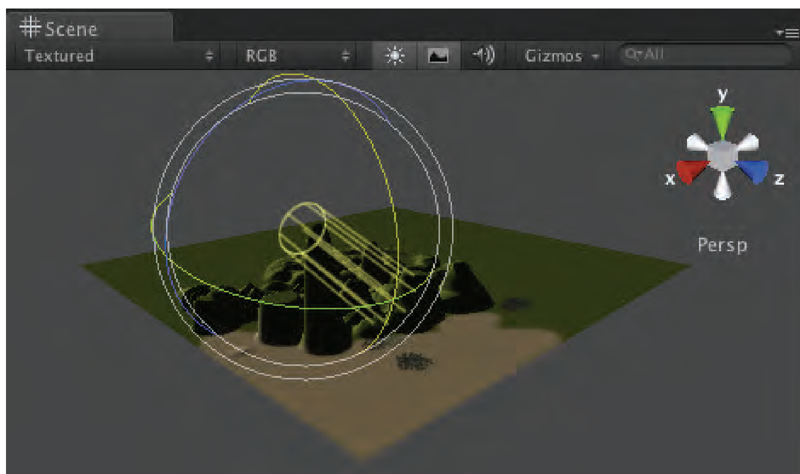


图5-29　旋转后的光源效果

5.4　天空盒子

在3D世界中，所有游戏元素都置身于天空盒子当中。天空没什么神秘的，读者可以将天空想象成一个巨大的盒子，这个盒子将整个游戏视图都包在了其中。在Unity引擎中制作天空盒子非常方便，只需简单几步就可以完成。

制作天空盒子之前，我们首先需要寻找天空的贴图资源。Unity为开发者提供了天空盒子资源包，里面包含很多天空的资源贴图，使用这些贴图可以制作一个美丽的天空。首先在Project视图中点击鼠标右键，从弹出的快捷菜单中选择“Import Package”→“Skyboxes”菜单项，将天空盒子资源包引入工程，如图5-30所示。在资源包中，共含有9款天空贴图资源。因为天空盒子由立方体组成，所以我们需要包含6个面的贴图材质。在右侧的Inspector视图中，可看到每一个天空材质共有六个面的贴图，分别是Front（前）、Back（后）、Left（左）、Right（右）、Up（上）和Down（下）。

5.4.1　Skybox组件

因为摄像机照射的面正是游戏显示的内容，所以可在摄像机上绑定一个Skybox组件，用于在Game视图中直接显示天空盒子贴图。首先在Hierarchy视图中选择Main Camera游戏对象（当前摄像机），然后在Unity导航菜单栏中选择“Component”→“Rendering”→“Skybox”菜单项，即可将天空盒子组件添加至主摄像机当中，如图5-31所示。然后在天空盒子组件中设置“Custom Skybox”（自定义天空盒子）贴图资源：点击右侧的按钮，程序将弹出选择天空盒子贴图界面，选择一款天空盒子材质添加至其中即可。运行游戏后，美丽的天空就出现在游戏画面中。

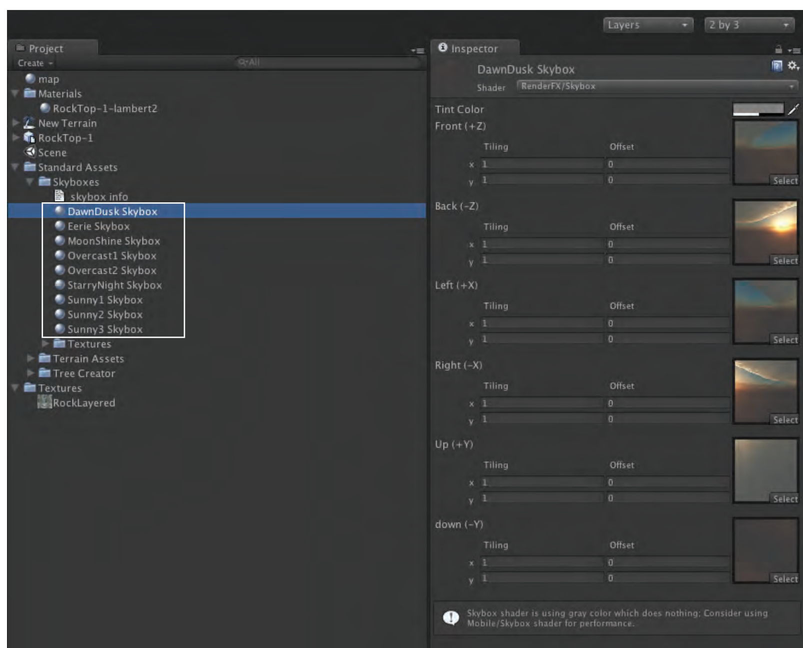


图5-30　天空资源包

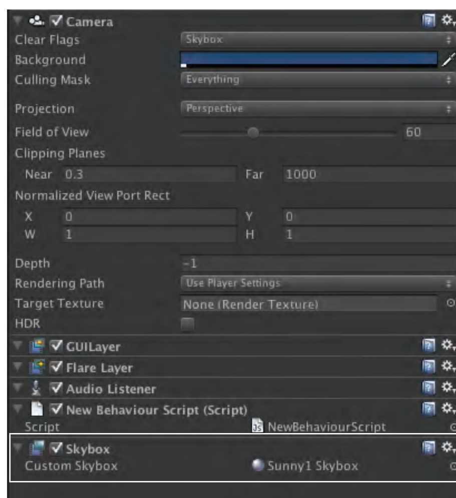


图5-31　摄像机

这里需要注意的是，如果游戏中只存在一个摄像机对象，这样添加天空盒子没有任何问题，但是如果游戏中存在多个摄像机对象，在摄像机之间进行切换后天空盒子贴图的位置就会出现问题，因为贴图资源只是相对于当前摄像机对象的，多个摄像机无法模拟同一个天空盒子。

5.4.2　在场景中添加天空盒子

在游戏场景中直接设置天空盒子，可避免在多个摄像机中设置天空盒子带来的切换视角后贴图显示的问题。在场景中添加天空盒子的方法如下：首先在Unity导航菜单栏中选择“Edit”→“Render Settings”菜单项，打开渲染设置界面（如图5-32所示），在该界面的“Skybox Material”（天空盒子材质）选项中设置天空的材质，然后将其直接应用于游戏场景中。如此添加的天空盒子就不必担心摄像机切换的视角问题，因为它是将真正的盒子放在游戏世界当中了。

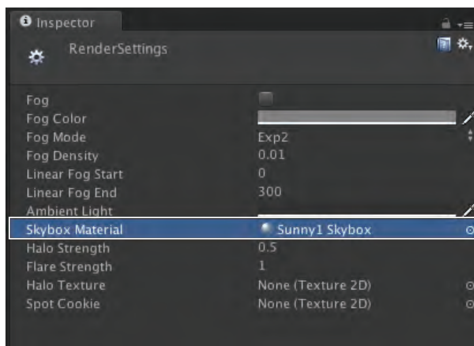


图5-32　渲染设置

运行游戏后，天空盒子的效果如图5-33所示。当然，读者也可自行设置天空盒子的材质，具体方法是在某材质中，在Shader（着色器）下拉列表中选择“RenderFx/SkyBox”，然后使用贴图资源将六面的贴图赋值填充即可，最后将这个材质赋给天空盒子即可。

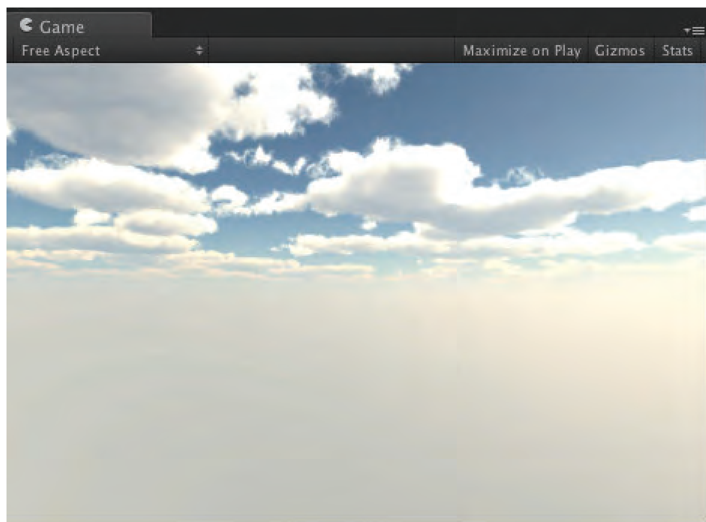


图5-33　天空盒子效果

5.5 常用编辑器组件

Unity游戏引擎的特色就是编辑器可视化，很多常用功能都可以在编辑器中完成。常用编辑器组件可分为两种，一是原有组件，二是拓展组件。原有组件是编辑器原生的一些功能，而拓展组件是编辑器之上通过脚本来拓展的新功能。本节中，我们将学习Unity中一些常用的编辑器组件。

5.5.1 摄像机

摄像机组件是Unity的核心组件之一，游戏界面中显示的一切内容正是场景中摄像机所照射的部分。作为一个游戏对象，摄像机存在Scene视图中，它可以设置自身的位置、照射的方向、照射的面积和照射的图层等。下面我们先来学习一下摄像机组件（如图5-34所示）的一些常用参数。

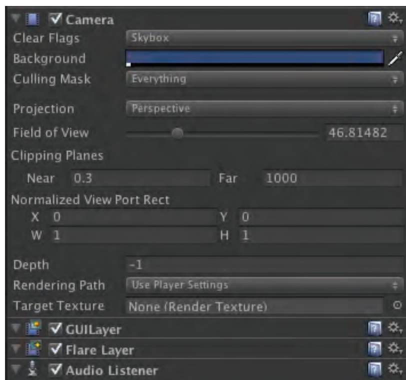


图5-34 摄像机组件

- Clear Flags：背景显示内容，默认是Skybox（天空盒子），前提是必须在“Render Settings”中设置过天空盒子材质。
- Background：背景显示颜色。如果没有设置天空盒子，将显示这个颜色。
- Culling Mask：用于选择是否显示某些层，默认为“Everything”（全部显示）。
- Projection：摄像机的类型。
- Field of View：摄像机的视野范围。
- Near：以摄像机为圆心，绘制最近点的距离。
- Far：以摄像机为圆心，绘制最远点的距离。
- Normalized View Port Rect：可理解为设定Game视图的显示区域参数。多台摄像机可以通过设置各自显示区域来分屏同时显示。
- Depth：摄像机的深度。若存在多个摄像机，先渲染该值较小的摄像机。
- Rendering Path：渲染路径。
- Target Texture：目标纹理，设置后会挡住摄像机。

5.5.2 摄像机的类型

Unity将摄像机分为两种类型，它们观察模型的角度是完全不同的：一种是放射观察角度，另一种是垂直观察角度。摄像机的类型将直接影响Game视图中的渲染效果，下面我们将学习如何设置摄像机的类型。在Hierarchy视图中选择摄像机对象，然后在右侧的Inspector视图中单击“Projection”下拉列表，如图5-35所示，可以看到该列表中有两个选项，分别是Perspective和Orthographic，从中选择其中一项后，即可将其应用到游戏当中。

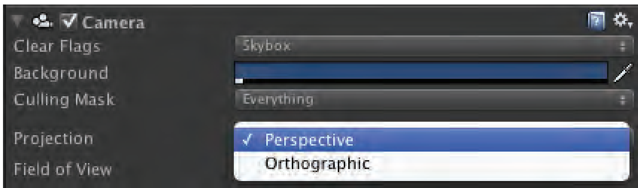


图5-35 摄像机类型

在“Projection”下拉列表中选择“Perspective”后摄像机的角度呈放射性观察，如图5-36所示。

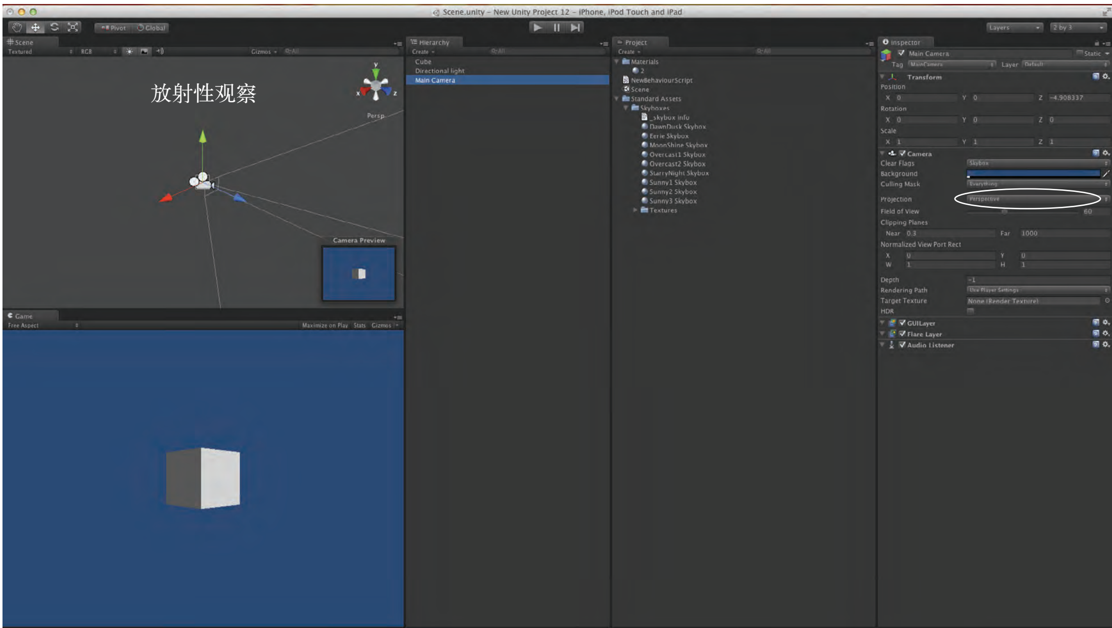


图5-36 放射性观察

在“Projection”下拉列表中选择“Orthographic”后摄像机的角度呈垂直性观察，如图5-37所示。

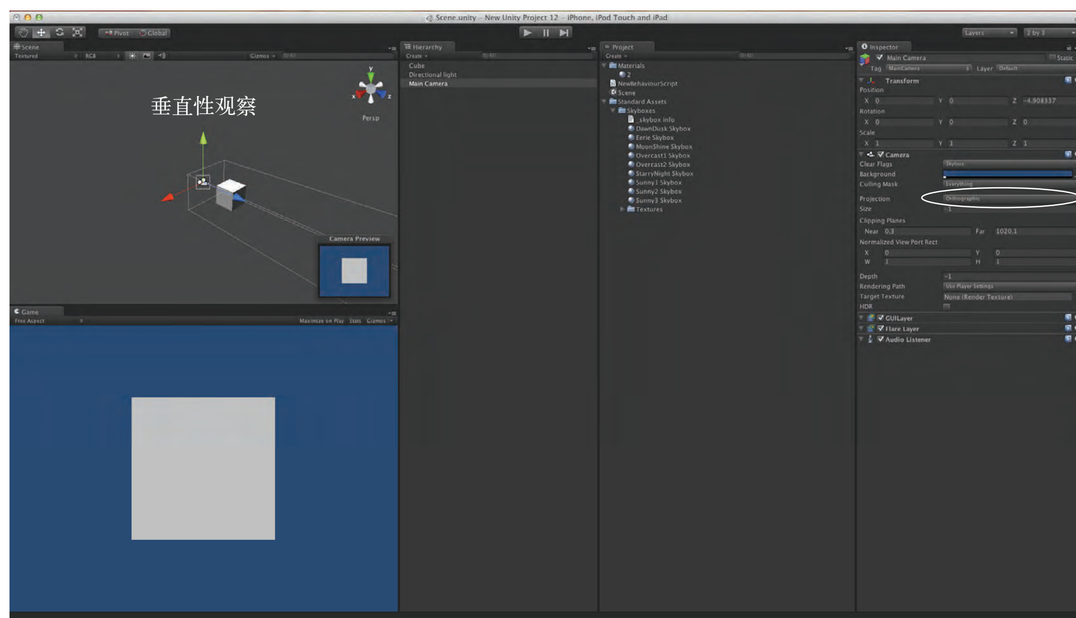


图5-37　垂直性观察

下面我们将学习如何在脚本中动态修改摄像机的类型，如图5-38所示。在Game视图中点击“放射观察”按钮或“垂直观察”按钮，此时将修改摄像机的类型，具体代码如代码清单5-1所示。

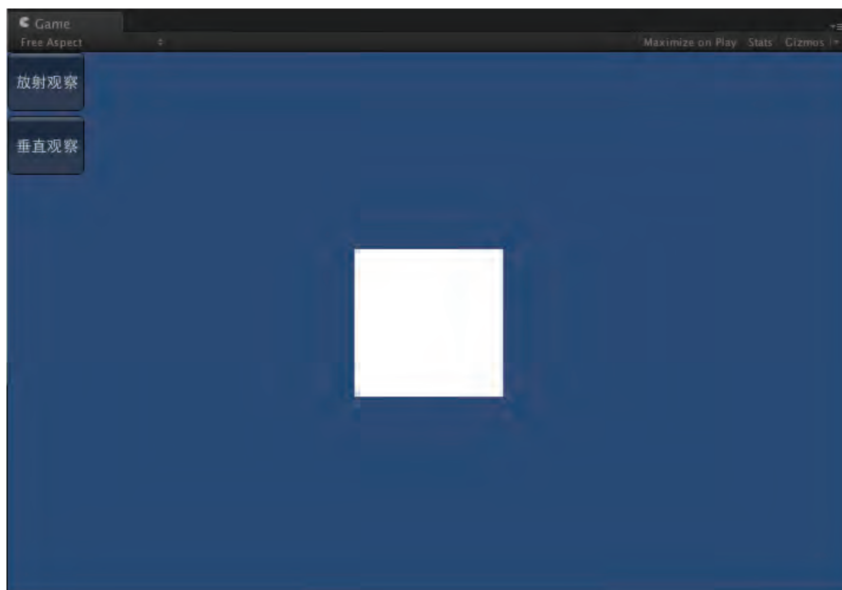


图5-38　摄像机动态观察

代码清单5-1 Script_05_02.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_05_02 : MonoBehaviour
{

    //摄像机组件
    private Camera camera;

    void Start ()
    {
        //获取摄像机组件
        camera = gameObject.GetComponent<Camera>();

    }

    void OnGUI()
    {
        if(GUILayout.Button("放射观察",GUILayout.Height(50)))
        {
            camera.isOrthoGraphic = true;
        }
        if(GUILayout.Button("垂直观察",GUILayout.Height(50)))
        {
            camera.isOrthoGraphic = false;
        }
    }
}
```

在上述代码中，我们使用isOrthoGraphic引用来修改当前摄像机的类型，将该变量赋值为true，表示摄像机的类型为垂直观察，赋值为false表示放射观察。

5.5.3 定制导航菜单栏

Unity导航菜单栏位于游戏引擎界面的顶部，其中有很多选项且含义各不相同。Unity为开发者提供了导航菜单栏的程序接口，使用代码可以动态添加菜单栏中的选项以及子项。本例在导航菜单栏中添加了一个“新的菜单栏”菜单项，如图5-39所示，具体代码如代码清单5-2所示。

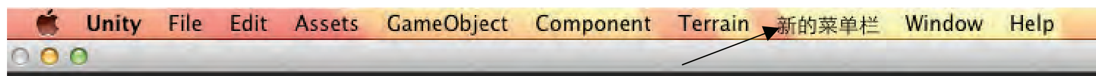


图5-39 新的菜单栏

代码清单5-2 Script_05_03.cs文件

```
using UnityEditor;
using UnityEngine;
class Script_05_03 : MonoBehaviour
{

    [MenuItem ("新的菜单栏/克隆选择的对象")]
```

```
    static void ClothObject()
    {
        Instantiate(Selection.activeTransform, Vector3.zero, Quaternion.identity);
    }

    [MenuItem ("新的菜单栏/克隆选择的对象", true)]
    static bool NoClothObject()
    {
        return Selection.activeGameObject != null;
    }

    [MenuItem ("新的菜单栏/删除选择的对象")]
    static void RemoveObject()
    {
        DestroyImmediate (Selection.activeGameObject,true);
    }

    [MenuItem ("新的菜单栏/删除选择的对象", true)]
    static bool NoRemoveObject()
    {
        return Selection.activeGameObject != null;
    }

}
```

在上述代码中，我们使用方法`[MenuItem ("新的菜单栏/克隆选择的对象")]`在导航菜单栏中添加一个菜单项，该方法的参数为菜单中子选项的完整路径。如图5-40所示，子菜单项已经添加至导航菜单栏中。选择“克隆选择的对象”或“删除选择的对象”菜单项，程序将会执行`MenuItem()`方法下面静态方法中的内容。

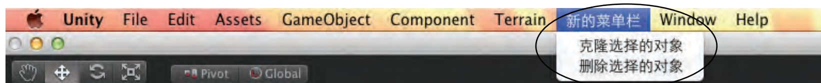


图5-40　编辑菜单栏

在Hierarchy视图中，选择一个游戏对象后，点击“克隆选择的对象”按钮或者“删除选择的对象”按钮，将进行相关操作。但是如果未选择游戏对象，直接点击“克隆选择的对象”按钮或者“删除选择的对象”按钮，程序将会出错，因为没有找到需要克隆与删除的对象。

我们可以使用`[MenuItem ("新的菜单栏/克隆选择的对象", true)]`方法来过滤选择对象，也就是说，如果未选择游戏对象，这两个按钮将呈灰色状态，如图5-41所示。

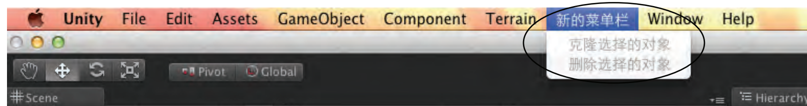


图5-41　无法选择

在导航菜单栏组件中，不仅可以绑定方法，还可以绑定整个脚本。由于方法不具备生命周期，所以它没脚本那么灵活。绑定脚本组件时，需要使用`AddComponentMenu()`方法。如图5-42所示，

本例将一个自动旋转的脚本添加至“Component”菜单项中，具体代码如代码清单5-3所示。



图5-42　脚本组件

代码清单5-3　Script_05_04.cs文件

```
using UnityEditor;
using UnityEngine;

//添加该脚本至“Component”菜单项中
[AddComponentMenu("新的脚本/自动旋转")]
class Script_05_04 : MonoBehaviour
{

    void Update()
    {
        //自身旋转
        transform.Rotate(0.0f,Time.deltaTime * 200,0.0f);
    }
}
```

添加脚本组件前，必须选择添加的游戏对象。如图5-43所示，如果未选择游戏对象，该按钮将呈无法选择状态。



图5-43　无法选择

5.5.4　预设

之前我们介绍过如何使用代码克隆游戏对象，预设与它的原理基本一样，只是表现方式不同而已。使用代码克隆对象适合在对象数量不确定时使用，而预设更适合在编辑器中编辑游戏场景时使用，适合对象数量已知的情况。

创建预设的方式如下，在Project视图中点击“Create”→“Prefab”菜单项即可，如图5-44所示。创建完预设后，是一个空对象，然后在Hierarchy视图中创建一个立方体对象，将其拖动至右侧创建的预设对象来为预设资源赋值。

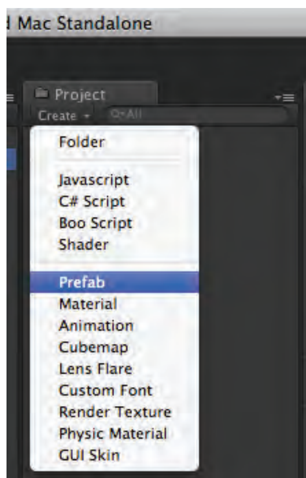


图5-44　创建预设

如图5-45所示，将预设文件拖动至Hierarchy视图中，即可完成对象的创建。这里可能会有朋友问它与创建一个立方体对象时有什么区别？使用预设可以减少内存开支，本例中使用预设创建的4个立方体对象共用一块立方体内存，而创建4个立方体对象好比它占用4块不同的立方体对象内存。

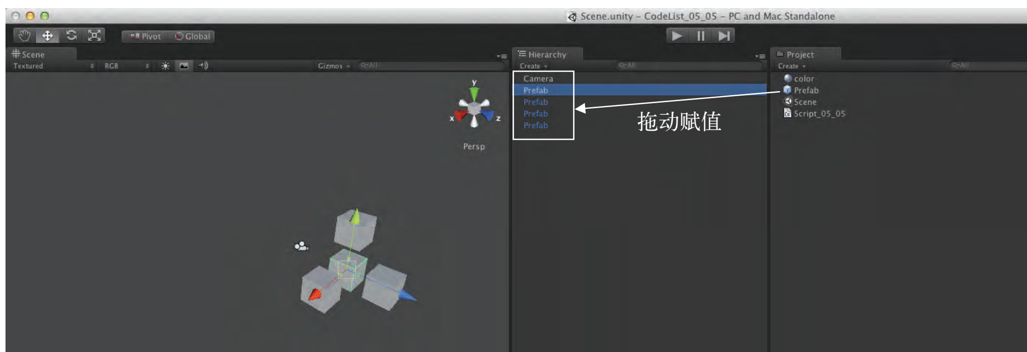


图5-45　使用预设

此外，使用预设还有一个重要的优势，那就是编辑预设资源后，场景中所有使用预设克隆后的游戏对象将全部适应新编辑的资源，无须一个一个给场景中的对象赋值。如图5-46所示，将红色的材质与自身旋转模型脚本赋值给预设资源，此时Hierarchy视图中预设克隆的对象材质全部变成红色，并且执行自身旋转的脚本。

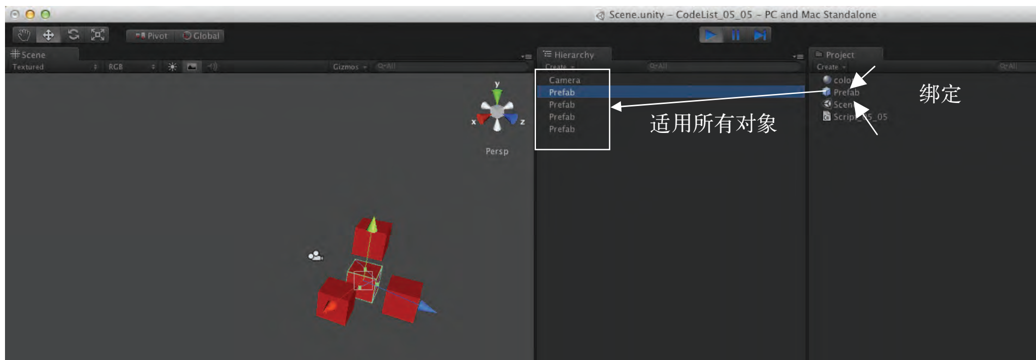


图5-46　使用预设

5.5.5　抗锯齿

在3D渲染过程中，可能会出现锯齿。如图5-47所示，仔细观察游戏视图中立方体的棱角处，可以发现锯齿现象非常明显。为什么会出现锯齿呢？因为它为了运行效率而牺牲了渲染效果，锯齿越多说明渲染效率就越快，锯齿越少说明渲染效率就越慢。

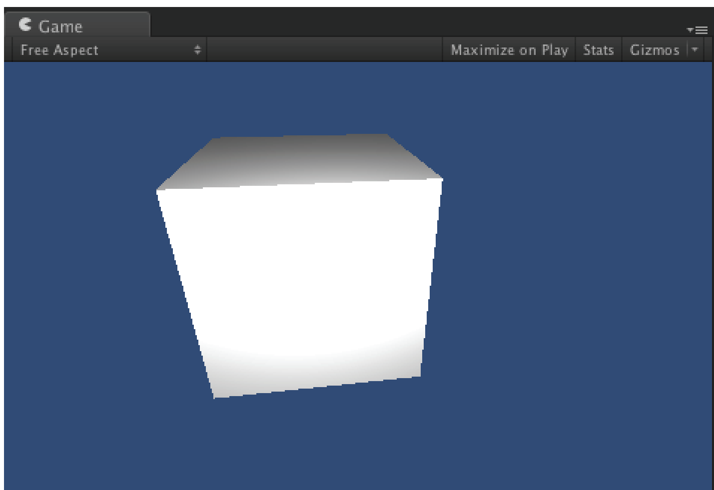


图5-47　锯齿

在Unity中可设置模型抗锯齿效果。在Unity导航菜单栏中选择“Edit”→“Project Settings”→“Quality”菜单项，此时右侧将弹出游戏质量设置窗口，如图5-48所示。根据不同游戏平台，共分了好几种渲染质量。Levels表示设置4个不同游戏平台的渲染效果，在对应平台下方的“Default”右侧点击小三角，此时将弹出渲染质量菜单，从中可直接选择该平台下渲染的效果等级。



图5-48 抗锯齿选项

默认情况下，渲染效果分为6个等级，它们是Fastest、Fast、Simple、Good、Beautiful和Fantastic。他们依次呈升序状态，Fastest表示运行速度最快，但是相对渲染质量会差一些，Fantastic表示运行速度最慢，但是渲染质量最好。点击下方的“Add Quality Level”按钮，可继续添加一个级别，不过需要我们自己手动去配置渲染质量中的所有参数，如图5-49所示。

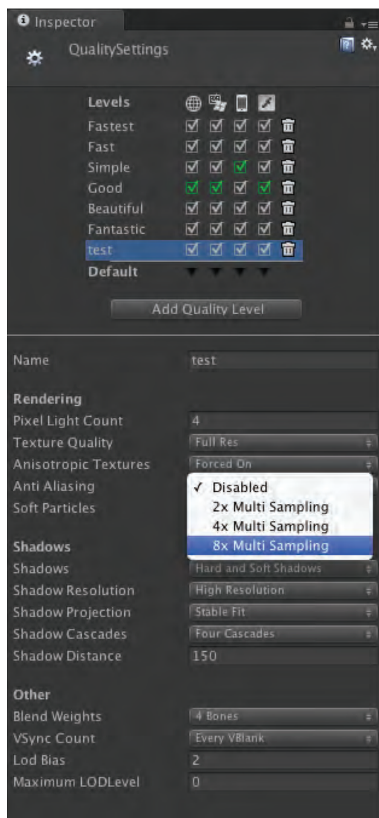


图5-49 渲染质量参数

点击“Add Quality Level”按钮来添加一个新的渲染质量等级，我们将它命名为“test”，在页面下方可看到渲染质量的相关参数，下面先简要介绍一下它们的含义。

- Name：渲染质量等级的名称。
- Pixel Light Count：渲染像素灯光的最大数量。
- Texture Quality：选择贴图质量，质量越好相对越消耗机器性能。
- Anisotropic Textures：贴图的显示等级。
- Anti Aliasing：设置抗锯齿的等级，其值越大表示抗锯齿的效果就越好。
- Soft Particles：是否开始混合粒子效果。
- Shadows：设置阴影的类型。
- Shadow Resolution：阴影的效果级别。
- Shadow Projection：选择阴影效果的分辨率。
- Shadow Cascades：设置阴影的层次等级。
- Shadow Distance：阴影的投射距离。
- Blend Weights：节点的重量。
- VSync Count：设置同步渲染数量。
- Lod Bias：LOD的偏移量。
- Maximum LODLevel：最大LOD等级。

在“Anti Aliasing”处修改抗锯齿级别为“8x Multi Sampling”，在游戏视图中看到的效果如图5-50所示，可以发现该立方体已经完成抗锯齿，棱角非常平滑完整。

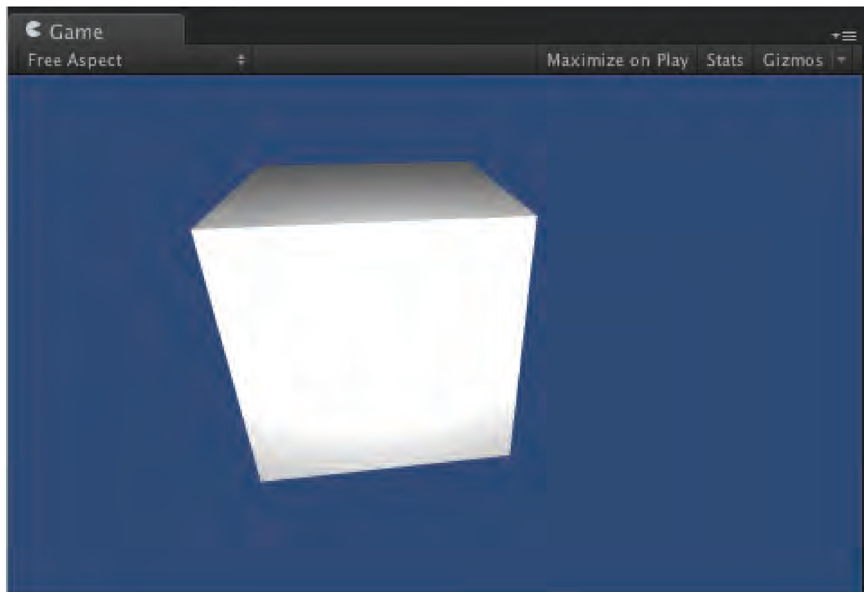


图5-50 抗锯齿效果

5.6　游戏实例——摄像机切换镜头

在游戏场景中，通常会存在多个摄像机，它们之间可相互切换来观察游戏中的某一个对象，这使得游戏画面更为逼真。

本例中我们将制作一个优美的游戏地形实例，效果如图5-51所示，正面效果如图5-52所示。在场景中分别创建3个游戏摄像机——以侧面、正面和上面三个方向照射游戏世界，点击任意按钮可切换摄像机观察的角度，具体代码如代码清单5-4所示。



图5-51　游戏地形效果图



图5-52　正面效果图

代码清单5-4　Script_05_07.js文件

```
private var Camera0 : GameObject;
private var Camera1 : GameObject;
private var Camera2 : GameObject;

function Start()
{
    //获取摄像机对象
    Camera0 = GameObject.Find("Camera0");
    Camera1 = GameObject.Find("Camera1");
    Camera2 = GameObject.Find("Camera2");
}

function OnGUI()
{

    if(GUILayout.Button("投射侧面",GUILayout.Height(50)))
    {
            //关闭Camera1与Camera2
            Camera1.active = false;
            Camera2.active = false;
            //打开Camera0
            Camera0.active = true;
    }

    if(GUILayout.Button("投射正面",GUILayout.Height(50)))
    {
        //关闭Camera0与Camera2
        Camera0.active = false;
        Camera2.active = false;
        //打开Camera1
        Camera1.active = true;

    }

    if(GUILayout.Button("投射上面",GUILayout.Height(50)))
    {
        //关闭Camera0与Camera1
        Camera0.active = false;
        Camera1.active = false;
        //打开Camera2
        Camera2.active = true;

    }
}
```

在上述代码中，在`Start()`方法中先获取3个摄像机对象，然后通过对象去调用`active`引用，将`active`设置为`true`时表示激活摄像机使其显示在屏幕中，设置为`false`表示取消该摄像机的显示。

5.7　本章小结

本章主要讲解游戏世界中的不常用元素，首先介绍了构建地形的方法与技巧，接着介绍了地形元素，如树木、草和石头等，然后讨论了3种不同的灯光类型——点光源、聚光灯和平行光，之后讲解了天空盒子的制作方法，接着学习了摄像机对象、脚本对象、预设对象和导航菜单栏中的动态添加与删除功能以及模型的抗锯齿功能，最后以一个游戏实例的形式向读者介绍如何在Unity场景中切换游戏摄像机。

第6章

物理引擎

物理引擎就是在游戏中模拟真实的物理效果。比如，场景中有两个立方体对象，一个在空中，一个在地面上，在空中的立方体开始自由下落，然后与地面上的立方体对象发生碰撞，而物理引擎就是用来模拟真实碰撞后的效果。

Unity的物理引擎使用的是NVIDIA（英伟达）的PhysX。目前，PC上很多游戏发烧友使用的显卡都是英伟达，该显卡是专门为游戏而设计的，它的物理引擎的效果很好，可以完美地在3D世界中模拟任何物理效果。该引擎完全可适用于次世代游戏开发，它渲染的游戏画面更加逼真，给玩家身临其境的感觉。如需让模型感应物理引擎的效果，需要将刚体组件或者角色控制器组件添加至该对象当中，刚体组件所受的物理效果是完美虚拟现实中的物理效果，而角色控制器需要受一些限制条件来感应物理效果。

6.1　刚体

刚体是一个非常重要的组件。新创建的物体默认情况下是不具备物理效果的，而刚体组件可以给物体添加一些常见的物理属性，比如物体质量、摩擦力和碰撞参数等，这些属性可用来真实模拟该物体在3D游戏世界中的一切行为。刚体可以以游戏组件的形式绑定在物体当中。如果物体添加了刚体组件，那么它将感应物理引擎中的一切物理效果。

6.1.1　简单使用

要想使用刚体，首先需要将刚体组件添加至游戏对象当中，具体操作方法如下：在Unity中创建一个需要添加刚体组件的游戏对象，比如立方体对象、球体对象、模型对象等含有网格的对象皆可，接着在Hierarchy视图中选择刚刚创建的游戏对象，然后在Unity导航菜单栏中选择“Component”→“Physics”→“Rigidbody”菜单即可。

下面我们将创建三个立方体，分别将其凌空放置在地面上，并且只给其中一个立方体添加刚体组件。运行游戏后，发现红色立方体正常感应了物理效果，从空中落了下来（因为只给它添加了刚体组件），而剩下的两个立方体依然停留在空中，如图6-1所示。

在Hierarchy视图中选择添加过刚体组件的游戏对象，此时在右侧的Inspector视图中可清晰地看到刚体组件包含的属性（如图6-2所示），下面简要介绍一下其中各个属性的含义。

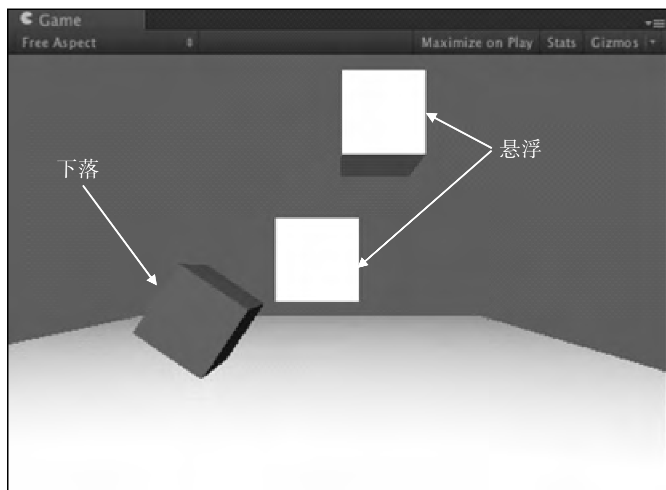


图6-1　物理下落

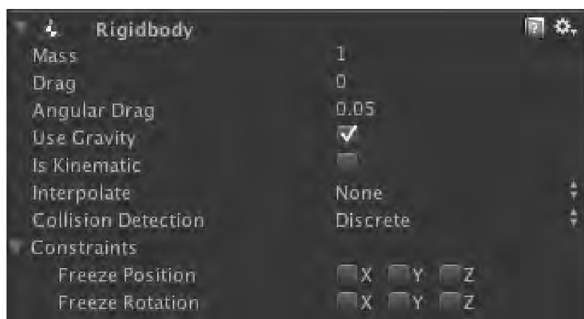


图6-2　刚体的属性

- Mass：质量，数值越大物体下落得也就越快。尽量不要让数值其超过10，否则物理效果就不会很真实。
- Drag：阻力，数值越大物体速度减慢得就越快。
- Angular Drag：角阻力，数值越大自身旋转的速度减慢得就越快。
- Use Gravity：是否使用重力。
- Is Kinematic：是否受物理的影响。
- Interpolate：设置图像差值。
- Collision Detection：碰撞检测。
- Constraints：冻结，停止某个轴向感应物理引擎的效果。
    - Freeze Position：冻结x轴方向、y轴方向和z轴方向。
    - Freeze Rotation：冻结x轴旋转、y轴旋转和z轴旋转。

6.1.2 物理管理器

在物理管理器中，可设置整个项目中所有物理效果的一些参数，比如默认物体的重力、反弹力、速度和角速度等。在导航菜单栏中选择“Edit”→“Project Settings”→“Physics”菜单项，此时即可打开“Physics Manager”（物理管理器）界面（如图6-3所示），下面简要介绍一下其中可设置的所有物理参数。

- Gravity：重力，默认情况下物体受y轴向下的重力为9.8N。可任意修改x轴、y轴和z轴3个方向的默认重力。
- Default Material：默认物理材质。
- Bounce Threshold：反弹值。如果一个小球从空中自然下落，下落到最低点时的速度低于反弹值，则不再向上反弹，保持为静止状态。
- Sleep Velocity：睡眠速度。当速度低于睡眠速度时，它保持静止状态。
- Sleep Angular Velocity：睡眠角速度。当角速度低于睡眠角速度时，自身不再旋转。
- Max Angular Velocity：最大角速度。
- Min Penetration For Penalty：用于物体与物体之间碰撞后的最小穿透力。
- Solver Iteration Count：迭代数量，默认值为7。
- Raycasts Hit Triggers：是否启动命中触发器。
- Layer Collision Matrix：图层碰撞矩阵。

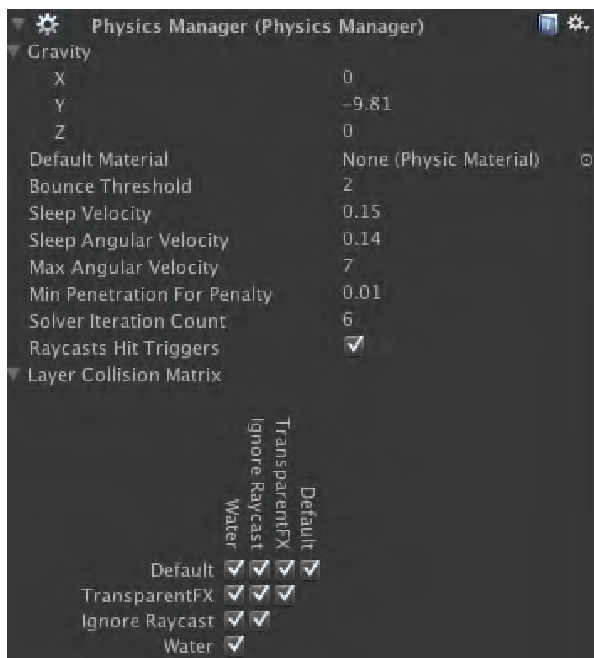


图6-3 “Physics Manager”界面

6.1.3　力

力在物理学中是一个非常重要的元素，其种类有很多。刚体组件可以受力的作用，比如给刚体施加一个*x*轴方向的力，那么该刚体绑定的物体将沿着*x*轴方向向前移动，这就好比用力将物体扔出去一样，该物体将会以抛物线的形式移动，而不是呆板地做匀速平移。

Unity中力的方式有两种：第一种为普通力，需要设定力的方向与大小；第二种为目标位置力，需要设定目标点的位置，该物体将朝向这个目标位置施加力。如图6-4所示，我们共放置两个球体对象，点击“普通力”按钮后，小球将像一脚被踢开似地向前滚动。点击“位置力”按钮后，小球将被施加一个朝向目标点的力，然后向目标滚动，本例的目标位置就是场景中的立方体，具体代码如代码清单6-1所示。

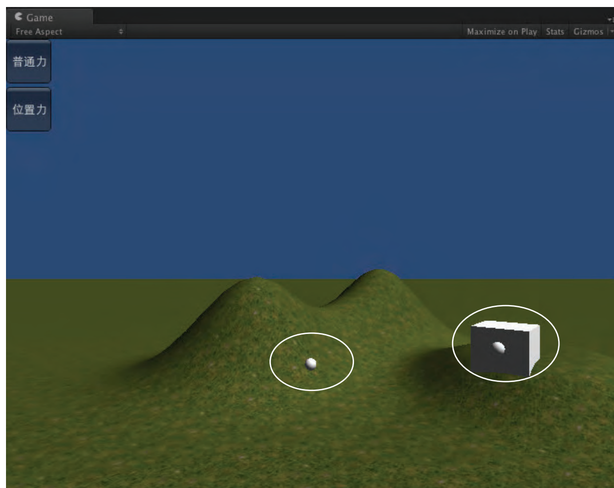


图6-4　力

代码清单6-1　Script_06_02.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_06_02 : MonoBehaviour
{

    //施加普通力的对象
    GameObject addFrceObj = null;
    //施加目标位置力的对象
    ameObject addPosObj = null;
    //目标对象
    GameObject cubeObj = null;

    void Start()
    {
```

```
        //获取施加普通力的对象
        addFrceObj = GameObject.Find("Sphere0");
        //获取施加目标位置力的对象
        addPosObj = GameObject.Find("Sphere1");
        //获取目标对象
        cubeObj = GameObject.Find("Cube");
    }

    void OnGUI()
    {
        if(GUILayout.Button("普通力",GUILayout.Height(50)))
        {
            //施加一个力，x轴方向力的大小为1000，y轴方向力的大小为1000
            addFrceObj.rigidbody.AddForce (1000, 0, 1000);
        }
        if(GUILayout.Button("位置力",GUILayout.Height(50)))
        {
            //施加一个位置力，物体将会向这个位置移动，力的模式为冲击力
            Vector3 force = cubeObj.transform.position - addPosObj. transform.position;
                addPosObj.rigidbody.AddForceAtPosition(force,addPosObj.transform.
                    position, ForceMode.Impulse);
        }
    }
}
```

在上述代码中，我们使用方法`rigidbody.AddForce()`添加一个普通力，该方法的参数是施加力的方向，单位是`Vector3`容器。使用方法`rigidbody.AddForceAtPosition()`施加一个目标位置力，该方法的第一个参数为目标点的位置坐标，第二个参数为力的模式。

6.1.4 碰撞与休眠

刚体与物体之间是存在碰撞的。一旦刚体开始移动，就可以在系统方法中监听到刚体的碰撞状态。刚体的碰撞可分为三种：进入碰撞、碰撞中和碰撞结束。关于休眠，可以将其理解为让模型变成静止状态，比如给一个正在运动的物体添加一个休眠状态，那么这个物体将马上静止，不再继续运动。下面先学习一下碰撞的3个重要的系统方法。

- `OnCollisionEnter()`：在刚体与刚体开始接触时，立刻调用此方法。
- `OnCollisionStay()`：在刚体与刚体碰撞的过程中，每帧都会调用此方法，直到碰撞结束。
- `OnCollisionExit()`：在刚体与刚体停止接触时，调用此方法。

如果有刚体与这个物体发生碰撞，程序立刻将碰撞游戏对象的详细信息显示在屏幕当中，并且碰撞结束后将碰撞的刚体设置成休眠状态，具体代码如代码清单6-2所示。

代码清单6-2 CollisionTest.cs文件

```
using UnityEngine;
using System.Collections;
```

```
public class CollisionTest : MonoBehaviour
{

    //碰撞显示信息
    string show = null;
    void Start()
    {
        //默认显示内容
        show = "未发生碰撞";
    }

    //进入碰撞
    void OnCollisionEnter(Collision collision)
    {
        show = "进入碰撞，碰撞名称：" + collision.gameObject.name;
    }
    //碰撞中
    void OnCollisionStay(Collision collision)
    {
        show = "碰撞中，碰撞名称：" + collision.gameObject.name;
    }
    //碰撞结束
    void OnCollisionExit(Collision collision)
    {
        show = "碰撞结束，碰撞名称：" + collision.gameObject.name;
        //碰撞结束后让物体休眠
        collision.gameObject.rigidbody.Sleep();

    }

    void OnGUI()
    {
        //将碰撞信息显示出来
        GUI.Label(new Rect(100,0,300,40),show);
    }
}
```

在上述代码中，`OnCollisionEnter()`、`OncollisionStay()`和`OnCollisionExit()`方法中的参数均为`Collision`对象，并且通过`collision.gameObject`即可得到当前碰撞的游戏对象。

6.2　碰撞器

游戏对象如果需要感应碰撞，那么必须给其添加碰撞器。默认情况下，创建游戏对象时，会自动将碰撞器组件添加至其中，而碰撞器组件决定了模型碰撞的方式。Unity一共为对象提供了5种碰撞器，分别是Box Collider（盒子碰撞器）、Sphere Collider（球体碰撞器）、Capsule Collider（胶囊碰撞器）、Mesh Collider（网格碰撞器）和Wheel Collider（车轮碰撞器），其中Box Collider适用于立方体对象之间的碰撞，Sphere Collider适用于球体对象之间的碰撞，Capsule Collider适用于胶囊体对象之间的碰撞，Mesh Collider比较特殊，它的碰撞方式由自定义模型的自身网格决定，

适用于自定义网格的碰撞，Wheel Collider适用于车轮与地面或其他对象之间的碰撞。

在碰撞器之间可以添加物理材质，用于设定物理碰撞后的效果。物理材质与碰撞器之间的关系非常紧密，比如两个立方体相撞后，按照物理引擎的效果，它们将开始相互反弹，那么反弹的力度就是由物理材质决定的。

6.2.1 添加碰撞器

添加碰撞器的方式如下：打开Unity，在Hierarchy视图中选择一个需要添加碰撞器的游戏对象，此时在Unity导航菜单栏中选择“Component”→“Physics”菜单项，然后从中选择碰撞器的种类，如图6-5所示。



图6-5 添加碰撞器

6.2.2 物理材质

物理材质可设定物体的表面材质，不同的表面材质可影响碰撞后的物理效果。物理材质可以模拟自然界中所有的物理碰撞后的效果，比如木头之间的碰撞或者冰块之间的碰撞。此外，物理材质可添加在任何碰撞器中，下面我们就在“Material”中添加碰撞器的物理材质，如图6-6所示。

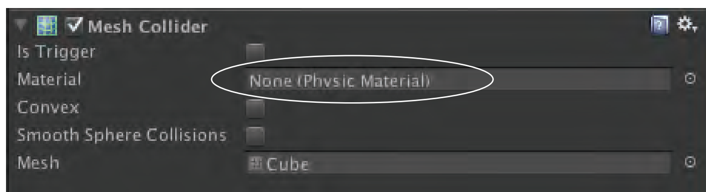


图6-6 添加物理材质

Unity标准资源包提供了一些物理材质的资源，我们可以将它们添加至当前工程中。在Project视图中单击鼠标右键，以弹出的快捷菜单中选择“Import Package”→“Physic Materials”菜单项，此时即可引入物理材质的标准资源包。

标准资源包中提供了5种常用的物理材质：弹性材质（Bouncy）、冰材质（Ice）、金属材质（Metal）、橡胶材质（Rubber）和木头材质（Wood）。如图6-7所示，首先在Scene视图中选择需要

添加物理材质的物体，然后在Project视图中通过拖动将Bouncy赋值给Inspector视图中的“Sphere Collider”即可。

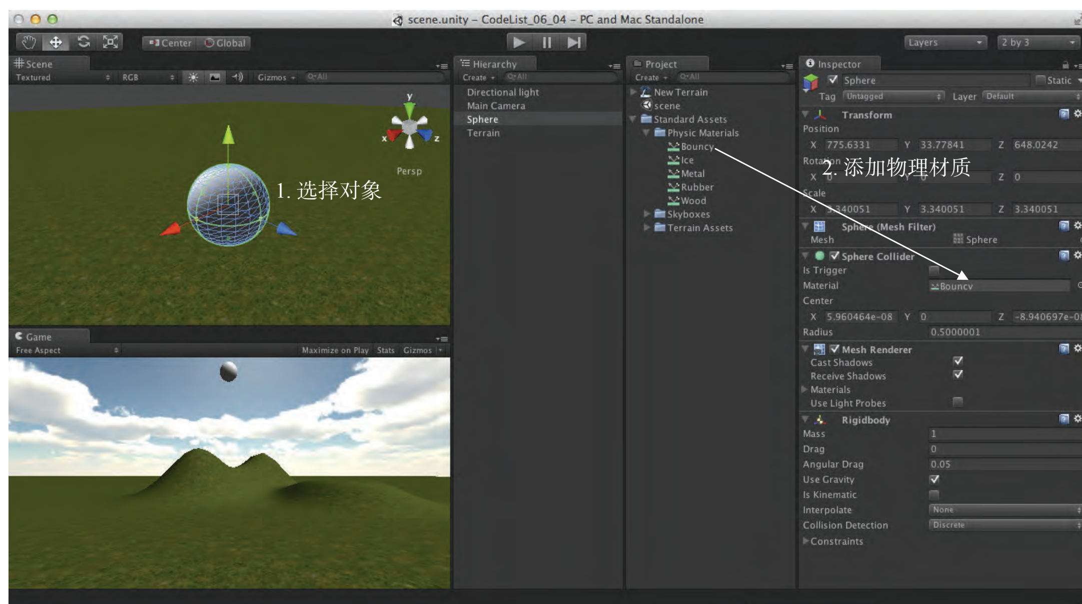


图6-7　添加物理材质

如图6-8所示，因为Game视图中的小球对象添加了弹性材质，运行游戏后，小球将从空中自然下落，落地后感应弹性材质，然后再次弹回至空中，如此循环下去。

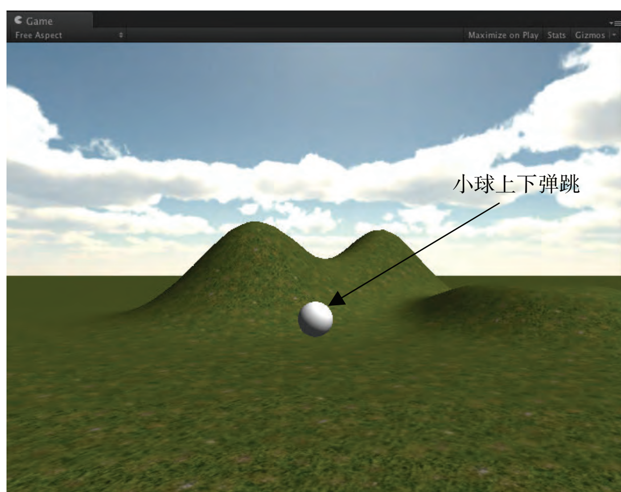


图6-8　弹性材质

除了可以使用Unity标准资源包中的5款物理材质，也可以自行创建物理材质。创建新物理材质的方法如下，在Project视图中点击“Create”→“Physic Material”菜单项即可。如图6-9所示，在右侧的Inspector视图中，我们将看到新创建的物理材质的所有属性，下面简要介绍一下这些属性的含义。

- Dynamic Friction：动态摩擦，取值在0到1之间。0表示最小动态摩擦，1表示最大动态摩擦。
- Static Friction：静态摩擦，取值在0到1之间。0表示最小静态摩擦，1表示最大静态摩擦。
- Bounciness：碰撞反弹系数，取值在0到1之间。0表示碰撞无反弹，1表示碰撞最大反弹。
- Friction Combine：普通碰撞后的摩擦模式。
- Bounce Combine：反弹碰撞后的摩擦模式。
- Friction Direction 2：摩擦方向，方向分为*x*轴、*y*轴和*z*轴。
- Dynamic Friction 2：动摩擦系数，它的摩擦方向根据Friction Direction 2设定。
- Static Friction 2：静摩擦系数，它的摩擦方向根据Friction Direction 2设定。

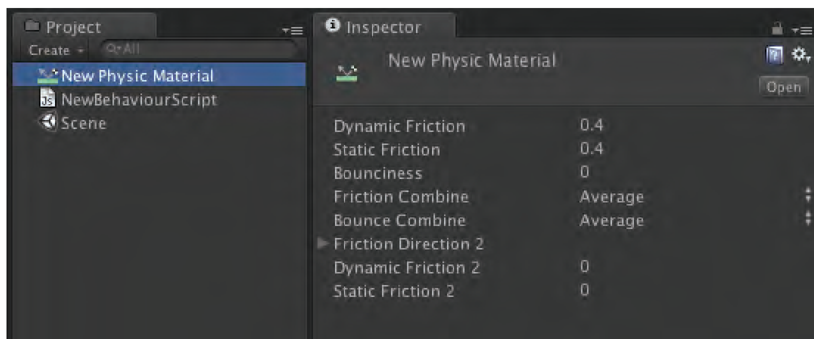


图6-9 自定义物理材质

6.3 角色控制器

Unity已经帮我们实现了“上”、“下”、“左”、“右”、“跳跃”等这些相对复杂的逻辑操作，并且将它们封装成了角色控制器组件，不用编写任何代码就可以轻松控制第一人称与第三人称主角。

角色控制器组件保存在Unity标准资源包中，它的功能非常强大，可模拟第一人称或第三人称视角。它不受刚体的限制，非常适用于表现游戏中的主角的运动。在使用角色控制器时，首先需要将它导入当前工程，具体操作方法为在Project视图中点击鼠标右键，从弹出的快捷菜单中选择“Import Package”→“Character Controller”菜单项即可。

6.3.1 第一人称

在第一人称视角游戏中，整个游戏视图好比主角的眼睛，游戏画面中的一切好像是从自己眼

睛看到的一样。第一人称视角的控制原理其实是控制Scene视图中摄像机对象的移动，所以屏幕显示的永远都是主角正前方的画面，比如经典的“CS”游戏就属于第一人称视角游戏。

将角色控制器组件导入工程后，在Project视图中找到该角色控制组件，然后将“First Person Controller”拖动至Hierarchy视图中即可，此时它将以一个胶囊体对象的形式出现在Scene视图中，然后使用移动工具编辑一下“胶囊”的位置，如图6-10所示。这里值得注意的是，第一人称视角组件的y轴一定要高于地面，否则运行游戏后，它将感应物理引擎坠落到地面之下。确保无误后直接运行游戏，第一人称视角遍映入我们眼帘，在Game视图中通过键盘按键“W”、“S”、“A”、“D”、“Space”可以直接控制第一人称视角的移动与跳跃。

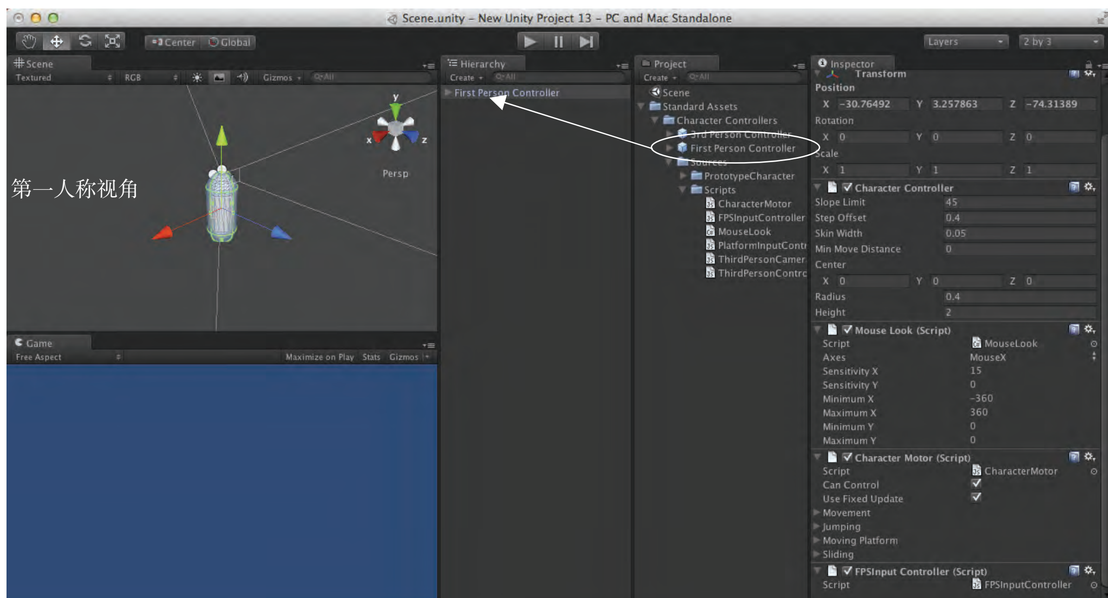


图6-10　添加第一人称组件

在Hierarchy视图中选择“First Person Controller”对象，在右侧的Inspector视图中可以看到这个对象共绑定了三条脚本，如图6-11所示。这3个脚本全部由官方提供，“MouseLook”脚本用来控制第一人称视角如何通过鼠标来移动整个视图，“CharacterMotor”脚本用来监听键盘事件，控制主角“前”、“后”、“左”、“右”的移动，“FPSInputController”脚本用来监听特殊的“Space”按键，实现第一人称视角的跳跃功能。三个脚本可以直接打开，所以也可在它们的基础上进行拓展，从而达到更好的控制效果。

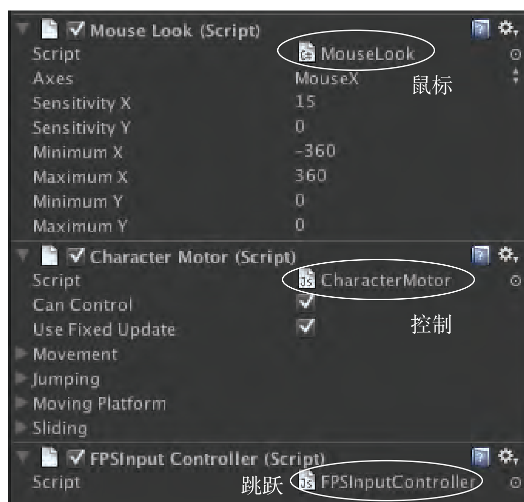


图6-11　“First Person Controller”对象绑定的3个脚本

6.3.2　第三人称

第三人称视角的原理是：在游戏场景中包含主角对象和摄像机对象，主角移动后，摄像机永远跟着主角移动，所以在Game视图中可以看出主角当前的移动方向。大多数游戏都采取第三人称的方式，因为第三人称更为清晰地向玩家展示控制角色的效果。下面将学习如何将第三人称视角添加至工程当中。如图6-12所示，首先在Project视图中将“3rd Person Controller”（第三人称控制器）拖动至Hierarchy视图中，然后在Scene视图中即可看到第三人称所用的模型对象。

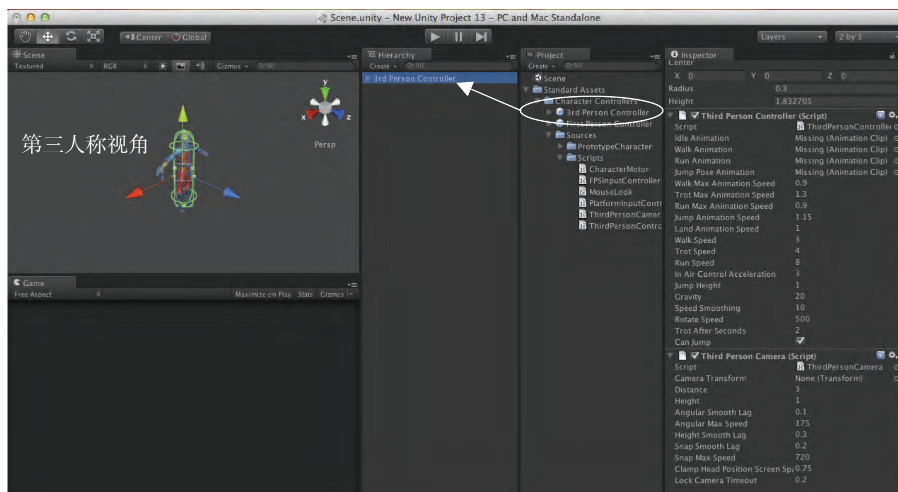


图6-12　第三人称

在Scene视图中加入地形，并且将角色控制器放在地形之上。运行游戏后即可看到第三人称视角的效果，用键盘的方向键控制主角移动，空格键用于控制主角跳跃。如图6-13所示，控制主角时，摄像机永远跟在主角身后，以第三人称的视角观看着主角的运动。



图6-13　运行游戏

6.3.3　控制组件

角色控制器组件和刚体组件都具备物理引擎的功能，它们都需要绑定在游戏对象中才能实现物理效果，并且同一个游戏对象中两者只能存在一个。刚体组件可以非常精确地模拟现实世界中模型的一切物理效果，而角色控制器则没有刚体那么精确，它更多地受控制器的限制，所以它更适合控制游戏对象。

举个简单的例子，给主角对象添加刚体组件作为物理引擎，控制主角向前快速移动，并且以较大力度碰撞在一面墙上。因为刚体组件会感应完全精确的物理效果，所以力度过大时，主角会被物理引擎施加的弹力反弹回来。因为主角虽然需要感应物理引擎的效果，但它是一个特殊的游戏对象，需要限制它的物理效果，此时就可以使用角色控制器组件来处理。

添加角色控制器之前，首先要确定是否把角色控制器标准资源包引入工程。如果未引入，则无法添加角色控制器组件。确保无误后，首先在Hierarchy视图中选择需要添加角色控制器的游戏对象，具体操作方法是在Unity导航菜单栏中选择“Component”→“Physics”→“Character Controller”菜单项即可，如图6-14所示。

添加完角色控制器组件后，就需要在脚本中控制角色控制器组件。首先创建一条游戏脚本，然后在脚本中使用方法`GetComponent<CharacterController>()`获取角色控制器组件对象，接着通过调用对象的`SimpleMove()`方法（其参数为角色移动的方向，只支持*x*轴与*z*轴方向的移动）来移动该角色。

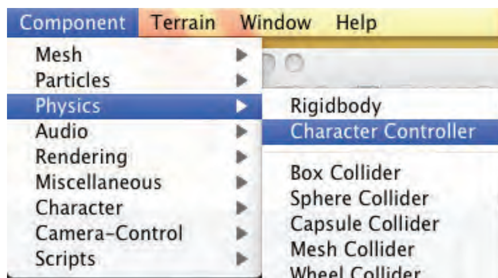


图6-14　添加控制器组件

如图6-15所示，本例通过角色控制器组件来控制立方体对象的旋转与移动，立方体正常感应物理引擎的碰撞，并且不会因为移动速度过快而被反弹回来，具体代码如代码清单6-3所示。

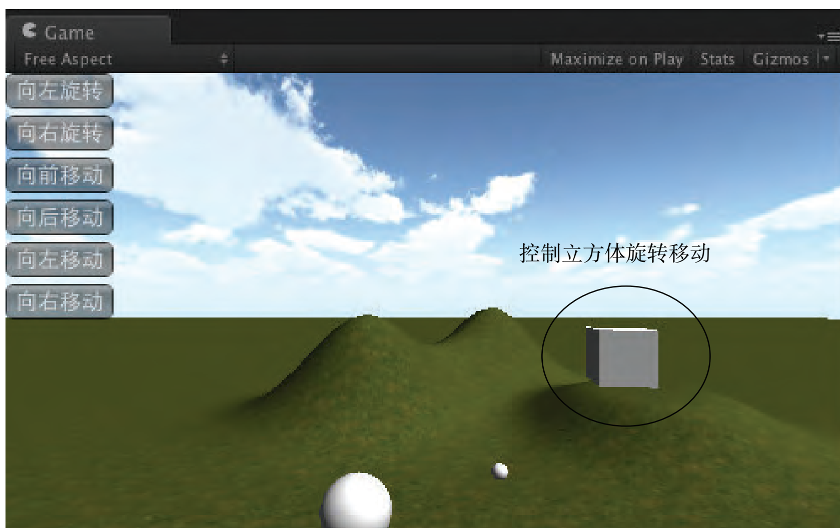


图6-15　控制移动

代码清单6-3　Script_06_05.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_06_05 : MonoBehaviour
{

    //角色控制器对象
    private CharacterController controller = null;
    //角色移动的速度
    private float moveSpeed = 30.0f;
    //角色旋转的速度
```

```
    private float rotateSpeed = 3.0f;

    void Start()
    {
        //获取角色控制器对象
        controller = GetComponent<CharacterController>();
    }

    void OnGUI()
    {
        //控制角色旋转
        if(GUILayout.RepeatButton("向左旋转"))
        {
            transform.Rotate(0,-rotateSpeed, 0);
        }
        if(GUILayout.RepeatButton("向右旋转"))
        {
            transform.Rotate(0,rotateSpeed, 0);
        }

        //控制角色移动
        if(GUILayout.RepeatButton("向前移动"))
        {
            controller.SimpleMove(Vector3.forward * moveSpeed);
        }
        if(GUILayout.RepeatButton("向后移动"))
        {
            controller.SimpleMove(Vector3.forward * -moveSpeed);
        }

        if(GUILayout.RepeatButton("向左移动"))
        {
            controller.SimpleMove(Vector3.right * -moveSpeed);
        }
        if(GUILayout.RepeatButton("向右移动"))
        {
            controller.SimpleMove(Vector3.right * moveSpeed);
        }
    }
}
```

6.3.4　移动与飞行

角色控制器组件可实现飞行与降落的功能，这可以通过`Move()`方法来实现，其中该方法的参数为飞行的角度。与`SimpleMove()`方法只适用于平面中的移动（也就是x轴与z轴之间的移动）不同，`Move()`方法适用于所有方向的移动，所以它可以实现游戏对象任意角度的飞行与移动效果。

在开发中，尽量使用`Move()`方法来控制角色移动，因为它完全可以取代`SimpleMove()`方法，并且使用它移动角色更为灵活。如图6-16所示，立方体可以实现起飞与降落的功能，具体代码如代码清单6-4所示。

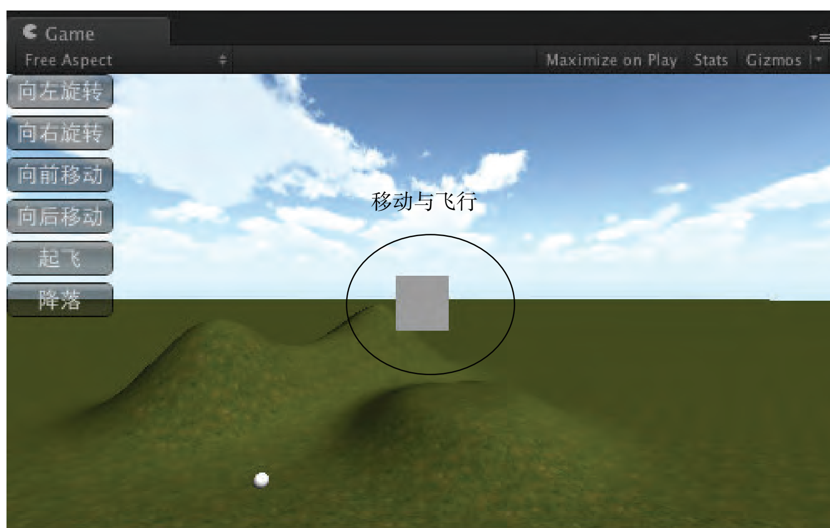


图6-16　控制飞行

代码清单6-4　Script_06_06.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_06_06 : MonoBehaviour
{

    //角色控制器对象
    private CharacterController controller = null;
    //角色移动的速度
    private float moveSpeed = 3.0f;
    //角色旋转的速度
    private float rotateSpeed = 3.0f;

    void Start()
    {
        //获取角色控制器对象
        controller = GetComponent<CharacterController>();
    }

    void OnGUI()
    {

        //控制角色旋转
        if(GUILayout.RepeatButton("向左旋转"))
        {
            transform.Rotate(0,-rotateSpeed, 0);
        }
        if(GUILayout.RepeatButton("向右旋转"))
```

6

```
        {
            transform.Rotate(0,rotateSpeed, 0);
        }

        //控制角色移动
        if(GUILayout.RepeatButton("向前移动"))
        {
            Vector3 forward = transform.TransformDirection(Vector3.forward);
            controller.Move(forward*moveSpeed);
        }
        if(GUILayout.RepeatButton("向后移动"))
        {
            Vector3 forward = transform.TransformDirection(Vector3.forward);
            controller.Move( forward*-moveSpeed);
        }

        //控制角色飞行与降落
        if(GUILayout.RepeatButton("起飞"))
        {
            transform.Translate(0, 1, 0);
        }
        if(GUILayout.RepeatButton("降落"))
        {
            transform.Translate(0, -1, 0);

        }
    }
}
```

在上述代码中，在控制主角移动时，可以使用transform.TransformDirection()方法得到当前主角移动时面朝的方向。

6.3.5　碰撞检测

角色控制器可感应游戏对象之间的碰撞，检测它们碰撞时，需要调用父类方法OnControllerColliderHit()。在该父类方法中，使用hit.gameObject引用，即可获取角色控制器组件碰撞后的游戏对象。如图6-17所示，本例控制立方体模型与平面游戏对象发生的碰撞，并且将碰撞对象的名称打印在屏幕中，具体代码如代码清单6-5所示。

代码清单6-5　Script_06_07.cs文件

```
using UnityEngine;
using System.Collections;
public class Script_06_07 : MonoBehaviour
{

    //角色控制器对象
    private CharacterController controller = null;
    //角色移动的速度
    private float moveSpeed = 3.0f;
    //角色旋转的速度
```

```
private float rotateSpeed = 3.0f;
//碰撞的游戏对象
private GameObject colliderObj = null;

void Start()
{
    //获取角色控制器对象
    controller = GetComponent<CharacterController>();
}

void OnGUI()
{

    //控制角色旋转
    if(GUILayout.RepeatButton("向左旋转"))
    {
        transform.Rotate(0,-rotateSpeed, 0);
    }
    if(GUILayout.RepeatButton("向右旋转"))
    {
        transform.Rotate(0,rotateSpeed, 0);
    }

    //控制角色移动
    if(GUILayout.RepeatButton("向前移动"))
    {
        Vector3 forward = transform.TransformDirection(Vector3.forward);
        controller.Move(forward*moveSpeed);
    }
    if(GUILayout.RepeatButton("向后移动"))
    {
        Vector3 forward = transform.TransformDirection(Vector3.forward);
        controller.Move( forward*-moveSpeed);
    }

    //控制角色的飞行与降落
    if(GUILayout.RepeatButton("起飞"))
    {
        transform.Translate(0, 1, 0);
    }
    if(GUILayout.RepeatButton("降落"))
    {
        transform.Translate(0, -1, 0);

    }

    //碰撞中
    if(controller.collisionFlags == CollisionFlags.Sides)
    {
        if(colliderObj != null)
        {
            GUI.color = Color.black;
            GUI.Label(new Rect(200,100,200,100),"碰撞的游戏对象为：" +
                colliderObj.name);
```

6

```
                }
            }
        }

        void OnControllerColliderHit(ControllerColliderHit hit)
        {
            //得到碰撞的游戏对象
            colliderObj = hit.gameObject;
        }
    }
```



图6-17　碰撞检测

6.4　射线

射线是3D世界中一个点向一个方向发射的一条无终点的线。在发射的轨迹中，一旦与其他模型发生碰撞，它将停止发射。我们可以判断某条射线是否发射至某游戏对象身上，其应用范围非常广，如射击类游戏中发射子弹后子弹行走的路径，并且通过这个路径可以判断子弹是否打中了目标物体。

6.4.1　射线的原理

创建一个射线时，首先需要知道射线的起点和终点在3D世界坐标系中的坐标。如图6-18所示，本例创建一条从3D世界零点射向游戏对象本身的射线，具体代码如代码清单6-6所示。

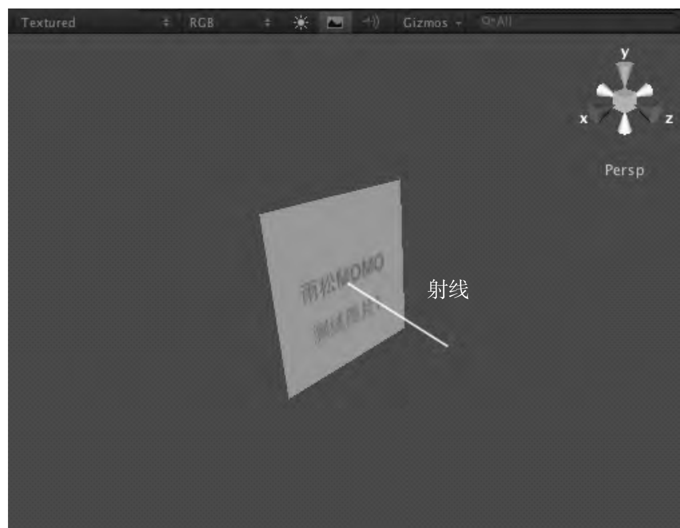


图6-18　创建的射线

代码清单6-6　Script_06_08.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_06_08 : MonoBehaviour
{

    void Update()
    {
        //创建射线，从零点发射到对象
        Ray ray = new Ray(Vector3.zero, transform.position);
        //计算射线的起点与终点
        RaycastHit hit;
        Physics.Raycast(ray, out hit, 100);
        //使用调试方法绘制这条线（调试方法仅在Scene视图中存在）
        Debug.DrawLine(ray.origin, hit.point);
    }

}
```

在上述代码中，`Debug.DrawLine()`方法只有在Scene视图中才能看到。如果想将射线绘制在游戏当中，需要使用GL图像库或者`LineRenderer()`方法，这在后面会详细说明。

6.4.2　碰撞检测

射线可用于判断与游戏对象的碰撞。如图6-19所示，本例以摄像机的位置为原点向鼠标移动点发射一条射线，好比向靶心打了一枪，这里这条射线用于判断是否打中在靶子上，具体代码如

代码清单6-7所示。

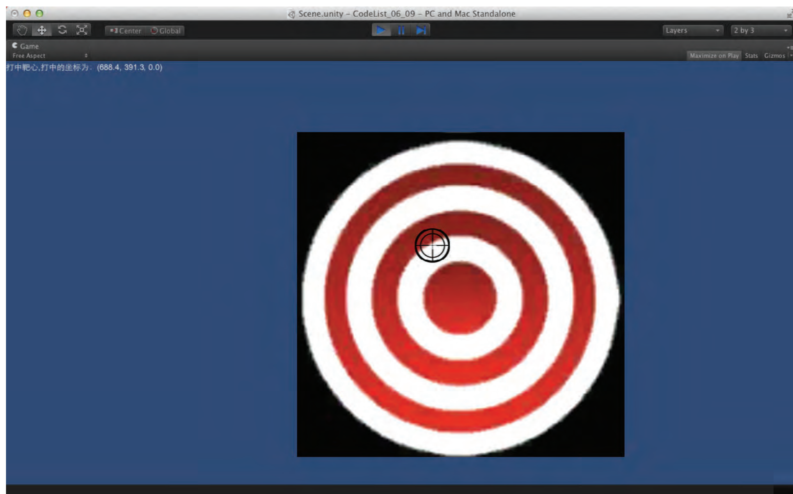


图6-19　射线碰撞

代码清单6-7　Script_06_09.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_06_09 : MonoBehaviour
{

    //靶心贴图
    public Texture texture;
    //提示信息
    private string info;

    void Update()
    {
        //创建从摄像机到鼠标之间的射线
        Ray ray =  Camera.main.ScreenPointToRay(Input.mousePosition);

        RaycastHit hit;

        //判断该射线是否打中游戏对象
        if(Physics.Raycast(ray, out hit))
        {
            info ="打中靶心";
        }else
        {
            info ="未打中靶心";
        }

    }
```

```
        void OnGUI()
        {
            //计算准心贴图的坐标
            Rect rect = new Rect(Input.mousePosition.x -(texture.width
                >>1 ) ,Screen.height - Input.mousePosition.y - (texture.height >>
                1),texture.width,texture.height);
            //绘制准心贴图
            GUI.DrawTexture(rect,texture);
            //输入打靶子的信息
            GUILayout.Label(info + ",打中的坐标为："+Input.mousePosition);
        }
    }
```

在上述代码中，我们使用 `Camera.main.ScreenPointToRay(Input.mousePosition)` 方法来创建一条由摄像机向鼠标当前位置发射的射线，然后使用`Physics.Raycast()`判断这条射线是否与某游戏对象相交，如果该方法返回`true`则表示相交，返回`false`则表示未相交。

6.5　关节

关节组件可添加至多个游戏对象当中，而添加了关节的游戏对象将通过关节连接在一起并且感应连带的物理效果，但是关节必须依赖于刚体组件。比如给两个游戏对象添加了弹性关节组件，这就好比给两个模型添加了一跟弹簧，当某一物体发生运动时，通过关节连带的另一个物体也将实现自由反弹效果。

6.5.1　关节介绍

关节组件一共分为5大类，它们分别是链条关节、固定关节、弹簧关节、角色关节和可配置关节，下面先简要介绍一下它们的具体含义。

- 链条关节（Hinge Joint）：将两个物体以链条的形式绑在一起，当力量过大超过链条的固定力矩时，两个物体就会产生相互的拉力。
- 固定关节（Fixed Joint）：将两个物体永远以相对的位置固定在一起，即使发生物理改变，它们之间的相对位置也不会发生改变。
- 弹簧关节（Spring Joint）：将两个物体以弹簧的形式绑定在一起，挤压它们会得到向外的推力，拉伸它们会得到两边对中间的拉力。
- 角色关节（Character Joint）：它可以模拟角色的骨骼关节，就好比人的手腕一样可以大范围任意角度旋转。
- 可配置关节（Configurable Joint）：它可以模拟任意关节的效果，包括上面的4种效果，它是最强大的，也是最复杂的。

关节是一个游戏组件，它既可以使用编辑器添加，也可以使用代码添加，本节先介绍使用编辑器添加的方法。如图6-20所示，在导航菜单栏中选择“Component”→“Physics”菜单项，然后从中选择一种关节组件，即可完成关节组件的添加。



图6-20　添加组件

小提示　可以设置关节的断裂。使用breakForce可设置关节断裂的力，一旦力度超过它，关节将会断裂。断裂时，可在OnJonitBreakForce(float breakForce)方法中监听到相关的事件。

6.5.2　实例——关节组件

如图6-21所示，本例将在两个立方体之间添加关节组件，并且当在Game视图中点击不同的按钮时，将实现两个立方体之间不同的关联关系，具体代码如代码清单6-8所示。

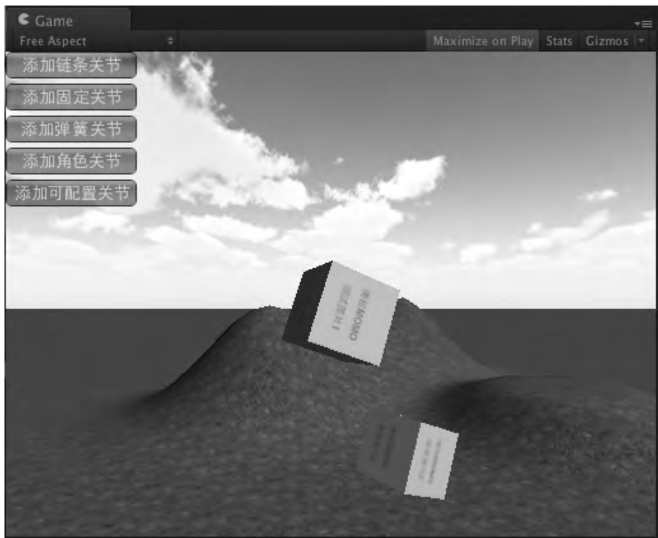


图6-21　关节实例

代码清单6-8 Script_06_10.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_06_10 : MonoBehaviour
{
    //链条关节游戏对象
    GameObject connectedObj = null;
    //当前链条关节组件
    Component jointComponent = null;

    void Start()
    {
        //获得链条关节的游戏对象
        connectedObj = GameObject.Find("Cube1");
    }

    void OnGUI()
    {
        if(GUILayout.Button("添加链条关节"))
        {

            ResetJoint();
            jointComponent = gameObject.AddComponent("HingeJoint");
            HingeJoint hjoint = (HingeJoint)jointComponent;
            connectedObj.rigidbody.useGravity = true;
            hjoint.connectedBody = connectedObj.rigidbody;
        }

        if(GUILayout.Button("添加固定关节"))
        {
            ResetJoint();
            jointComponent =gameObject.AddComponent("FixedJoint");
            FixedJoint fjoint = (FixedJoint)jointComponent;
            connectedObj.rigidbody.useGravity = true;
            fjoint.connectedBody = connectedObj.rigidbody;
        }

        if(GUILayout.Button("添加弹簧关节"))
        {
            ResetJoint();
            jointComponent =gameObject.AddComponent("SpringJoint");
            SpringJoint sjoint = (SpringJoint)jointComponent;
            connectedObj.rigidbody.useGravity = true;
            sjoint.connectedBody = connectedObj.rigidbody;
        }

        if(GUILayout.Button("添加角色关节"))
        {
            ResetJoint();
```

6

```
            jointComponent =gameObject.AddComponent("CharacterJoint");
            CharacterJoint cjoint = (CharacterJoint)jointComponent;
            connectedObj.rigidbody.useGravity = true;
            cjoint.connectedBody = connectedObj.rigidbody;
        }

        if(GUILayout.Button("添加可配置关节"))
        {
            ResetJoint();
            jointComponent =gameObject.AddComponent("ConfigurableJoint");
            ConfigurableJoint cojoint = (ConfigurableJoint)jointComponent;
            connectedObj.rigidbody.useGravity = true;
            cojoint.connectedBody = connectedObj.rigidbody;
        }
    }

    //重置关节
    void ResetJoint(){
        //销毁之前添加的关节组件
        Destroy (jointComponent);
        //重置对象位置
        this.transform.position = new Vector3(821.0f,72.0f,660.0f);
        connectedObj.gameObject.transform.position = new Vector3(805.0f,48.0f,660.0f);
        //不感应重力
        connectedObj.rigidbody.useGravity = false;
    }
}
```

本例中我们通过点击按钮来添加关节组件，每次添加关节组件时，都需要抹去之前添加的关节组件并且还原游戏对象的原始位置，然后使用AddComponent()方法将新的关节组件添加至游戏对象当中。

6.6　粒子特效

粒子效果的原理是将若干粒子无规则地组合在一起，用来模拟火焰、爆炸、水滴、雾气等效果。在Unity引擎中，使用粒子特效非常方便。

在学习粒子特效之前，先学习一下如何创建粒子对象，具体操作方法是在Hierarchy视图中点击“Create”→“Particle System”菜单项即可。创建好粒子对象后，即可在Scene视图中编辑该粒子对象的位置。

6.6.1　粒子发射器

粒子发射器用于设定粒子的发射属性，比如粒子的大小、数量和速度等。在Hierarchy视图中选择刚刚创建的粒子对象，在右侧的Inspector视图中可看到粒子发射器的所有属性，如图6-22所示。下面简要介绍一下各个属性的含义。

- ❑ Emit：是否使用粒子发射器。若不选中该选项，则不产生粒子特效。
- ❑ Min Size：粒子的最小尺寸。
- ❑ Max Size：粒子的最大尺寸。
- ❑ Min Energy：粒子的最小生命周期，单位为秒，意思为*N*秒后该粒子消失。
- ❑ Max Energy：粒子的最大生命周期，单位为秒，意思为*N*秒后该粒子消失。
- ❑ Min Emission：粒子每秒生成的最小数量。
- ❑ Max Emission：粒子每秒生成的最大数量。
- ❑ World Velocity：粒子在3D世界中各个轴的速度。
- ❑ Local Velocity：粒子自身坐标系中各个轴的移动速度。
- ❑ Rnd Velocity：各个轴粒子的随机速度。
- ❑ Emitter Velocity Scale：粒子继承发射的速度。
- ❑ Tangent Velocity：粒子发射切线的速度。
- ❑ Angular Velocity：粒子发射的角速度。
- ❑ Rnd Angular Velocity：粒子的随机角速度。
- ❑ Rnd Rotation：粒子是否随机旋转。
- ❑ Simulate in Worldspace：是否在世界坐标系中模拟粒子。
- ❑ One Shot：选中该选项后，粒子将只发射一次，否则粒子将连续发射。
- ❑ Ellipsoid：粒子产生的所有轴的位置。
- ❑ Min Emitter Range：设定粒子之间的间隙。



图6-22 粒子发射器

6.6.2 粒子动画

粒子动画用于设定粒子渲染中的动画效果。动画效果的种类繁多，比如动画的颜色、拉伸的

比例、拉伸的速度等，如图6-23所示。下面先简要介绍一下粒子动画中各个属性的含义。

- Does Animate Color：是否开启粒子动画的颜色，颜色将根据自身的生命周期发生改变。
- Color Animation[]：设置动画渐变数组中的颜色，这个数组的长度是5，也就是说粒子颜色发生改变时，循环的是数组中的这5种颜色。
- World Rotation Axis：粒子围绕世界坐标轴旋转。
- Local Rotation Axis：粒子围绕着本地空间轴旋转。
- Size Grow：粒子成长的生命周期。
- Rnd Force：粒子运行时，每经过一帧在粒子上施加一个随机力。
- Force：粒子运行时，每经过一帧在粒子上施加一个固定力。
- Damping：阻力，用于减慢粒子。
- Autodestruct：自动销毁粒子动画对象。

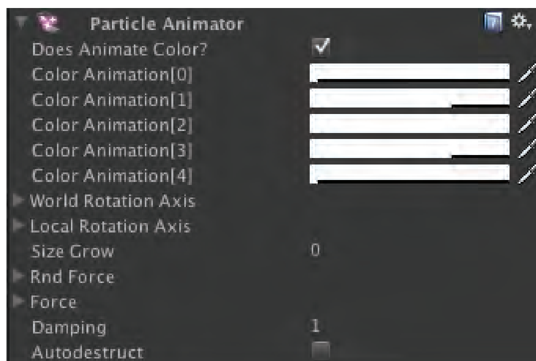


图6-23　粒子动画

6.6.3　粒子渲染器

粒子渲染器主要用于粒子的渲染，比如粒子渲染模式、粒子的缩放、粒子的尺寸等，如图6-24所示。下面简要介绍一下粒子渲染器中各个选项的具体含义。

- Cast Shadows：是否投射粒子的阴影。
- Receive Shadows：是否接收粒子的阴影。
- Materials：粒子显示的材质。
- Camera Velocity Scale：相机缩放的速度。
- Stretch Particles：粒子的显示状态，如横向、纵向等。
- Length Scale：粒子缩放的长度。
- Velocity Scale：粒子缩放的速度。
- Max Particle Size：粒子最大的尺寸。
- UV Animation：设置粒子动画水平方向上的数量与垂直方向上的数量以及播放贴图动画。

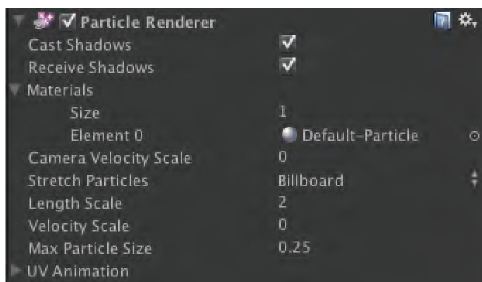


图6-24　粒子渲染器

6.6.4　粒子效果实例

Unity系统提供了粒子标准资源包，其中含有很多非常棒的粒子资源。在Project视图中单击鼠标右键，以弹出的快捷菜单中选择“Import Package”→“Particles”菜单项，将粒子标准资源包引入当前工程。资源包中有很多现成的粒子材质，本例从中选择一个名称为“Sparkles1”的材质。如图6-25所示，粒子特效映入眼帘，根据右侧的拖动条可动态修改资源特效的相关属性，具体代码如代码清单6-9所示。



图6-25 粒子效果

代码清单6-9　Script_06_11.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_06_11 : MonoBehaviour
{
```

```
//粒子对象
GameObject particle = null;
//粒子在x轴方向的速度
float velocity_x = 0.0f;
//粒子在y轴方向的速度
float velocity_y = 0.0f;
//粒子在z轴方向的速度
float velocity_z = 0.0f;

void Start()
{
    //获得粒子对象
    particle = GameObject.Find("ParticleSystem");

}

void OnGUI()
{
    //通过拖动设置粒子的最大尺寸
    GUILayout.Label("粒子最大尺寸");
    particle.particleEmitter.maxSize = GUILayout.HorizontalSlider
        (particle.particleEmitter.maxSize, 0.0f, 10.0f,GUILayout.Width(150));

    //通过拖动设置粒子的最大消失时间
    GUILayout.Label("粒子消失时间");
    particle.particleEmitter.maxEnergy = GUILayout.HorizontalSlider
        (particle.particleEmitter.maxEnergy, 0.0f, 10.0f,GUILayout.Width(150));

    //通过拖动设置粒子的最大生成数量
    GUILayout.Label("粒子的最大生成数量");
    particle.particleEmitter.maxEmission = GUILayout.HorizontalSlider (particle.
        particleEmitter.maxEmission, 0.0f, 100.0f,GUILayout.Width(150));

    //通过拖动设置粒子x轴的移动速度
    GUILayout.Label("粒子x轴的移动速度");
    velocity_x= GUILayout.HorizontalSlider (velocity_x, 0.0f, 10.0f,
       GUILayout.Width(150));
    particle.particleEmitter.worldVelocity = new Vector3(velocity_x,
        particle.particleEmitter.worldVelocity.y, particle.particleEmitter.
        worldVelocity.z);

    //通过拖动设置粒子y轴的移动速度
    GUILayout.Label("粒子y轴的移动速度");
    velocity_y= GUILayout.HorizontalSlider (velocity_y, 0.0f,
        10.0f,GUILayout.Width(150));
    particle.particleEmitter.worldVelocity = new
        Vector3( particle.particleEmitter.worldVelocity.x,velocity_y,
        particle.particleEmitter.worldVelocity.z);

    //通过拖动设置粒子z轴的移动速度
    GUILayout.Label("粒子z轴的移动速度");
```

```
            velocity_z= GUILayout.HorizontalSlider (velocity_z, 0.0f,
                10.0f,GUILayout.Width(150));
            particle.particleEmitter.worldVelocity = new
                Vector3( particle.particleEmitter.worldVelocity.x,
                particle.particleEmitter.worldVelocity.y,velocity_z);

        }
    }
```

在上述代码中，首先在Start()方法中使用Find()方法获取当前粒子效果的游戏对象，然后继续调用particleEmitter引用拿到当前粒子效果的发射器对象，最后根据发射器对象来编辑粒子效果。

6.6.5　布料

布料是Unity 3.x引入的特色组件。布料是柔软的，它可以变成任意形状，比如随风飘扬的旗子或窗户上的窗帘等。

创建布料的方式有两种：第一种为创建布料对象，第二种为在游戏对象中添加布料组件。前者通过在Hierarchy视图中选择“Create”→“Cloth”（布料）即可完成创建。创建完毕后，系统会自动将互动布料组件（Interactive Cloth）与布料渲染组件（Cloth Renderer）添加至该对象中。后者的实现方式是，在Unity导航菜单栏中选择“Component”→“Physics”→“Interactive Cloth”菜单项即可完成创建，如图6-26所示。



图6-26　交互布料组件

交互布料组件是由网格组成的布料，主要用于布料的逻辑判断，应用于摩擦、密度、气压等影响布料的效果，它们会影响到布料的具体物理数值之间的判断。布料渲染可以给布料绘制一张贴图，使其更加美观。皮肤布料（Skinned Cloth）比较特殊，主要用来模拟人物模型皮肤的布料，比如角色的衣服、裤子、帽子等。这些布料会根据角色骨骼动画的运动而发生相应的改变，所以角色布料更加细腻。

如图6-27所示，交互布料组件可设置若干参数，具体如下所示。

- Bending Stiffness：硬度，取值范围为0~1。
- Stretching Stiffness：韧性，取值范围为0~1。
- Damping：阻力，取值范围为0~1。
- Thickness：厚度，直接影响布料的质量大小。
- Use Gravity：使用重力。取消勾选该参数，布料将不受重力影响。
- Self Collision：自身碰撞。勾选该参数后，运算效率会大幅增加。
- External Acceleration：作用于布料的一个外力，它可以影响布料的默认行为。
- Random Acceleration：随机外力，比如所风飘动的旗子受到两个方向的随机外力。
- Mesh：网格面，决定布料的形状。
- Friction：摩擦力，取值范围为0 ~ 1。
- Density：密度，数值越大布料的质量就越高。
- Pressure：气压。
- Collision Response：与其他模型碰撞后的反馈，碰撞后布料将接收反馈的力度。
- Attachment Tear Factor：附带撕破系数。
- Attachment Response：附带反馈。
- Tear Factor：撕破系数，数值越大布料越不容易被撕破。
- Attached Colliders：附带碰撞器。

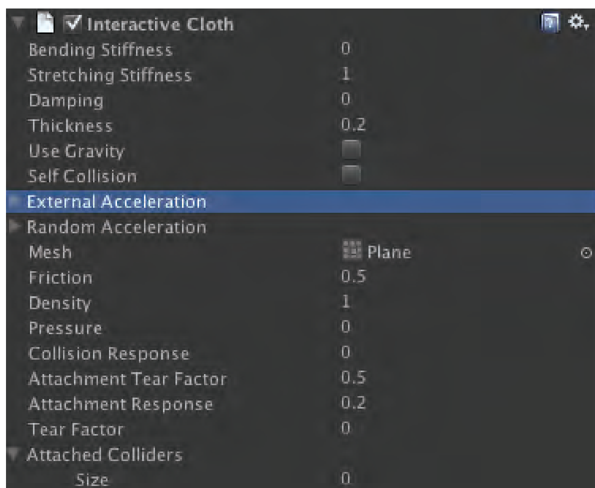


图6-27　组件参数

如图6-28所示，本例将布料组件添加至平面对象上，点击左侧的按钮可以控制平面对象移动的方向。由于布料与立方体对象发生了碰撞，所以布料发生了变形。本例代码如代码清单6-10所示。



图6-28　移动布料

代码清单6-10　Script_06_12.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_06_12 : MonoBehaviour
{

    //布料对象
    Cloth cloth = null;

    void Start()
    {
        //获取布料对象
        cloth = (Cloth)GetComponent<InteractiveCloth>();
    }

    void OnGUI()
    {
        //移动布料
        if(GUILayout.RepeatButton("向上"))
        {
            cloth.externalAcceleration =  new Vector3(0,1,0);
        }
```

```
        if(GUILayout.RepeatButton("向下"))
        {
            cloth.externalAcceleration =  new Vector3(0,-1,0);

        }
        if(GUILayout.RepeatButton("向左"))
        {
            cloth.externalAcceleration =  new Vector3(1,0,0);
        }
        if(GUILayout.RepeatButton("向右"))
        {
            cloth.externalAcceleration =  new Vector3(-1,0,0);

        }
    }

}
```

在上述代码中，我们首先在`Start()`方法中使用`GetComponent()`方法获取布料的组件对象，然后继续使用`externalAcceleration`引用给布料添加一个方向力，让其进行“向上”、“向下”、“向左”和“向右”的移动。

6.6.6 路径渲染

路径渲染属于特效渲染组件，用于跟随运动中的游戏对象。下面将学习如何将路径渲染组件添加至游戏中。首先在Hierarchy视图中创建一个球体对象，然后在导航菜单栏中选择“Component”→“Effects”→“Trail Renderer”菜单项，即可将路径渲染组件添加至该球体对象当中。

在Hierarchy视图中选择刚刚添加过路径渲染组件的球体对象，此时右侧的Inspector视图将显示路径渲染组件的所有参数，如图6-29所示。下面我们将学习这些参数的具体含义。

- ❑ Cast Shadows：显示阴影效果。
- ❑ Receive Shadows：接收阴影效果。
- ❑ Materials：材质。
  - ■ Size：渲染的材质数量，可添加或删除材质。
  - ■ Element 0：渲染材质的文件。
- ❑ Use Light Probes：是否使用光线探头。
- ❑ Light Probe Anchor：光线探头的参照物。
- ❑ Time：路径渲染的消失时间，跟随物体超过一段时间时则渐渐消失。
- ❑ Start Width：起始的宽度。
- ❑ End Width：结束的宽度。
- ❑ Colors：添加渲染渐变的颜色数组。
- ❑ Min Vertex Distance：最小顶点的距离。
- ❑ Autodestruct：自动销毁路径渲染对象。

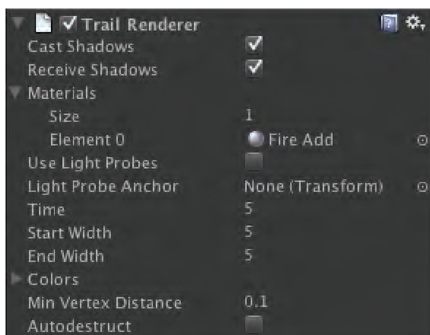


图6-29　路径渲染组件

下面我们将学习如何在脚本中应用路径渲染组件。为了让刚刚创建的球体对象有一个滚动的轨迹，继续给该对象添加刚体组件并且将其放在地形上。运行游戏后，该球体将感应物理引擎由空中自然下落。如图6-30所示，在Game视图中，点击“增加宽度”按钮，路径轨迹将逐渐变宽，具体代码如代码清单6-11所示。

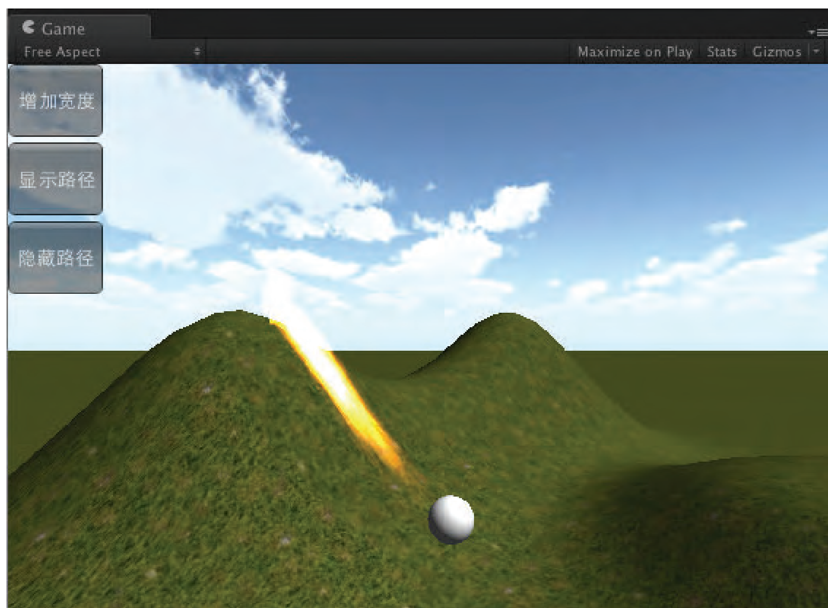


图6-30　路径渲染

代码清单6-11　Script_06_13.cs文件

```
using UnityEngine;
using System.Collections;
```

```
public class Script_06_13 : MonoBehaviour
{
    //路径渲染对象
    private TrailRenderer trialRender;

    void Start()
    {
        //获取路径渲染对象
        trialRender = gameObject.GetComponent<TrailRenderer>();
    }

    void OnGUI()
    {

        if(GUILayout.Button("增加宽度",GUILayout.Height(50)))
        {
            trialRender.startWidth +=1;
            trialRender.endWidth   +=1;
        }

        if(GUILayout.Button("显示路径",GUILayout.Height(50)))
        {
            trialRender.enabled = true;
        }

        if(GUILayout.Button("隐藏路径",GUILayout.Height(50)))
        {
            trialRender.enabled = false;
        }
    }
}
```

在上述代码中，我们首先在Start()方法中使用GetComponent()方法获取路径渲染组件的对象，然后通过修改startWidth与endWidth引用来修改路径渲染的宽度，接着通过enabled引用表示是否显示路径。

6.7　游戏实例——击垮围墙

在这个实例中，我们将利用物理引擎的知识，在游戏中发射炮弹以便击垮面前的围墙，如图6-31所示。首先在Game视图中选择炮弹发射目标点，然后单击鼠标左键向目标发射炮弹，最后待炮弹撞到围墙上时，围墙受物理引擎的影响而被击垮。

本例给炮弹与围墙均添加了刚体组件，给炮弹绑定了粒子系统，并且在炮弹的粒子动画中添加了5组不同的颜色，保持粒子特效将始终跟随炮弹的移动轨迹。选择发射目标后，将向该炮弹施加一个目标方向力，使其向目标点发射，具体代码如代码清单6-12所示。

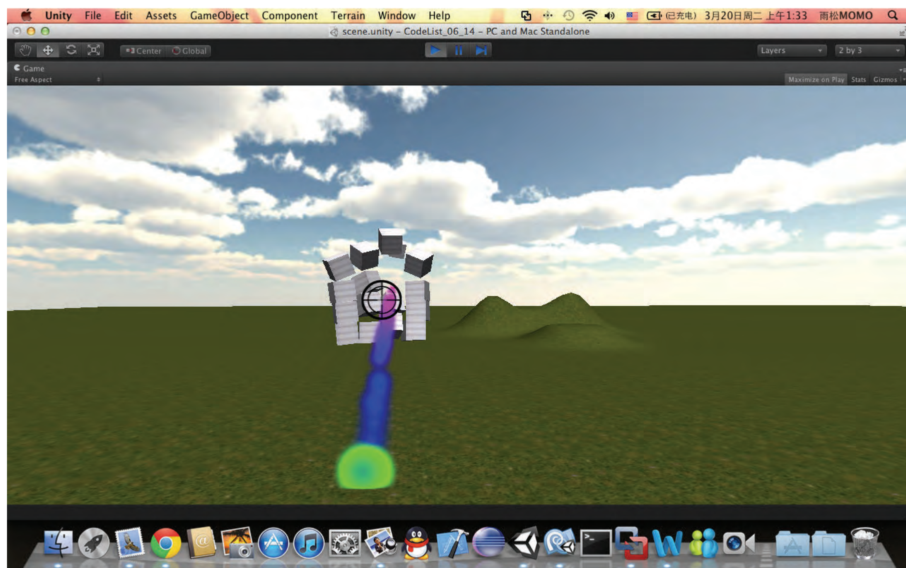


图6-31　击垮围墙

代码清单6-12　Script_06_14.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_06_14 : MonoBehaviour {

    //炮弹对象
    private GameObject obj;
    //准心贴图
    public Texture texture;

    void Start ()
    {
        //获取炮弹对象
        obj = GameObject.Find("Sphere0");
        //隐藏默认鼠标图标
        Screen.showCursor = false;

    }
    void FixedUpdate()
    {
        //点击鼠标左键后
        if(Input.GetMouseButton(0))
        {
            //创建从摄像机发射到鼠标位置的射线
            Ray ray =  Camera.main.ScreenPointToRay(Input.mousePosition);
            RaycastHit hit;
            //判断射线是否与游戏对象相交
```

```
            if(Physics.Raycast(ray, out hit))
            {
                //确保游戏对象为围墙
                if(hit.collider.name =="Cube")
                {
                    //计算炮弹与目标点之间的距离
                    Vector3 direction = hit.transform.position -
                        obj.transform.position;
                    //发射炮弹
                    obj.rigidbody.AddForceAtPosition(direction,
                        hit.transform.position,ForceMode.Impulse);
                }

            }
        }
    }

    void OnGUI()
    {
        //绘制准心
        Rect rect = new Rect(Input.mousePosition.x -(texture.width >>1 ) ,
            Screen.height - Input.mousePosition.y - (texture.height >>
            1),texture.width,texture.height);

        GUI.DrawTexture(rect,texture);

    }
}
```

本例中我们依照射线的原理，发射炮弹时，以摄像机为原点，向目标点发射了一条射线，并且确保目标位置在围墙上，然后使用`AddForceAtPosition()`方法向该目标点发射炮弹。

6.8　本章小结

本章首先介绍了如何将刚体组件应用到游戏对象中、如何给对象施加一个力、物理管理器以及碰撞器的相关内容，接着介绍了角色控制器的相关属性，然后讨论了射线、关节与粒子特效的使用方法，盘点了粒子组件的所有参数，最后通过给炮弹施加一个力使它击垮面前的围墙，向读者充分诠释了Unity物理引擎的强大功能。

第7章

输入与控制

输入与控制操作在游戏开发中会频繁使用，比如玩家控制主角移动、按键后攻击敌人、选择行走路线等行为都需要在程序中监听玩家在终端上输入的信息。Unity是一款跨平台的游戏引擎，其输入方式在不同平台上也会不太一样，比如在PC或Mac上通过键盘和鼠标输入，在Android或iPhone上通过触摸屏操作，所以在程序中需要根据不同的平台监听不同的控制事件。Unity为开发者提供了`Input`这个类库，其中包括键盘事件、鼠标事件和触摸事件等一切跨平台所需要的控制事件。Unity支持的平台数量过多，平台之间的控制方式差异较大，本章中我们将主要讲解PC平台下的基础事件。

7.1 键盘事件

键盘事件是PC最基本的输入方式之一。键盘按键的种类繁多，一般的PC都会有104个不同的按键。在程序中，可以通过监听键盘按键事件，从而执行一些逻辑，比如在经典的“CS”游戏中，“W”键表示向前移动，“S”键表示向后移动，“A”键表示向左移动，“D”键表示向右移动，“Space”键表示跳跃，鼠标左键表示开枪，右键表示放大准心等。在游戏开发中，往往要监听键盘按下事件、键盘抬起事件、键盘长按事件和键盘连按事件等，下面我们简要介绍一下这些事件。

7.1.1 按下事件

在脚本中，我们在`Input.GetKeyDown()`方法中将按键值作为参数传入即可判断该按键是否被按下，如果按键被按下，该方法将返回`true`，没有按下则返回`false`。如图7-1所示，本例共监听了键盘上的5个按键，分别是“W”、“S”、“A”、“D”和空格键。按下相应按键后，可以使用`Debug.Log()`方法将按键信息打印出来，具体代码如代码清单7-1所示。

代码清单7-1 Script_07_01.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_07_01 : MonoBehaviour
{
```

```
    void Update()
    {
        if (Input.GetKeyDown (KeyCode.W))
        {
            Debug.Log("您按下了W键");
        }

        if (Input.GetKeyDown (KeyCode.S))
        {
            Debug.Log("您按下了S键");
        }

        if (Input.GetKeyDown (KeyCode.A))
        {
            Debug.Log("您按下了A键");
        }

        if (Input.GetKeyDown (KeyCode.D))
        {
            Debug.Log("您按下了D键");
        }

        if (Input.GetKeyDown (KeyCode.Space))
        {
            Debug.Log("您按下了空格键");
        }
    }
}
```

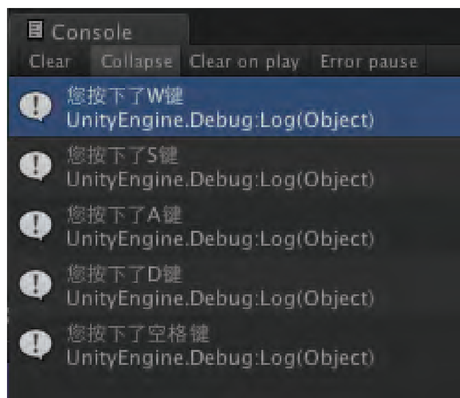


图7-1　按下事件

运行游戏后，直接在键盘上按下“W”键、“S”键、“A”键、“D”键或者空格键，即可在调试信息输出窗口中将按下的按键名称打印出来。

7.1.2 抬起事件

抬起事件的产生完全依赖于按下事件，因为只有键盘执行按下事件后，系统才会调用键盘抬起事件。在代码中，我们使用`Input.GetKeyUp()`方法得到某按键的抬起事件，该方法的参数为按键的键值。按键抬起后，该方法返回`true`，否则返回`false`。如图7-2所示，通过打印的信息，我们可以清晰地看到每一个按下事件与抬起事件的相互关系，具体代码如代码清单7-2所示。

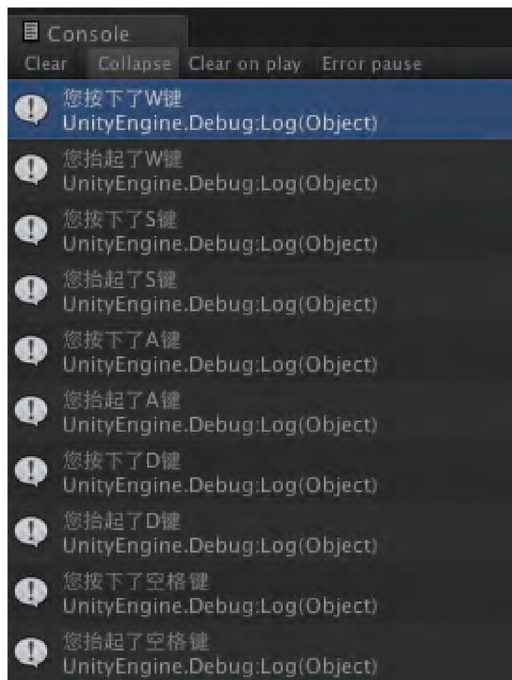


图7-2 按下与抬起事件

7

代码清单7-2 Script_07_02.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_07_02 : MonoBehaviour
{

    void Update()
    {
        //按下事件
        if (Input.GetKeyDown (KeyCode.W))
        {
            Debug.Log("您按下了W键");
        }
```

```
        if (Input.GetKeyDown (KeyCode.S))
        {
            Debug.Log("您按下了S键");
        }

        if (Input.GetKeyDown (KeyCode.A))
        {
            Debug.Log("您按下了A键");
        }

        if (Input.GetKeyDown (KeyCode.D))
        {
            Debug.Log("您按下了D键");
        }

        if (Input.GetKeyDown (KeyCode.Space))
        {
            Debug.Log("您按下了空格键");
        }

        //抬起按键
        if (Input.GetKeyUp (KeyCode.W))
        {
            Debug.Log("您抬起了W键");
        }

        if (Input.GetKeyUp (KeyCode.S))
        {
            Debug.Log("您抬起了S键");
        }

        if (Input.GetKeyUp (KeyCode.A))
        {
            Debug.Log("您抬起了A键");
        }

        if (Input.GetKeyUp (KeyCode.D))
        {
            Debug.Log("您抬起了D键");
        }

        if (Input.GetKeyUp (KeyCode.Space))
        {
            Debug.Log("您抬起了空格键");
        }

    }
}
```

运行游戏后，按下键盘上的“W”键、“S”键、“A”键、“D”键或者空格键后，松开按键后即可在调试信息输出窗口中打印键盘按下与抬起的按键名称。

7.1.3 长按事件

键盘长按事件可监听键盘中某个按键是否一直处于按下状态，比如在飞行射击类游戏中，玩家长按开火键时，子弹会一直处于发射状态。在代码中，我们可以使用`Input.GetKey()`方法判断键盘中某个按键是否一直处于按下状态。长按事件示例的代码如代码清单7-3所示。

代码清单7-3 Script_07_03.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_07_03 : MonoBehaviour
{
    //记录某按键按下的帧数
    int keyFrame = 0;


    void Update()
    {

        if (Input.GetKeyDown (KeyCode.A))
        {
            Debug.Log("A按下一次");
        }
        if (Input.GetKey (KeyCode.A))
        {
            //记录按下的帧数
            keyFrame++;
            Debug.Log("A连按:" + keyFrame+"帧");
        }
        if (Input.GetKeyUp (KeyCode.A))
        {
            //抬起后清空帧数
            keyFrame=0;
            Debug.Log("A按键抬起");
        }
    }
}
```

为了更清晰地观察键盘长按事件，本例在程序中声明了一个整型变量`keyFrame`来记录键盘长按下的时间，使用`Input.GetKey()`方法记录按下的时间帧数，然后直接将时间帧数打印出来。

7.1.4 任意键事件

在程序中，还可监听键盘中任意按键是否被按下。在常见游戏中，读取完资源后，会提示玩家按任意键继续操作，它的实现原理就是在操作界面中用程序监听终端任意键是否被按下。代码清单7-4就是用来监听任意键是否被按下，并且将按下的时间帧数打印在屏幕当中。

代码清单7-4　Script_07_04.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_07_04 : MonoBehaviour
{

    //记录某按键按下的帧数
    int keyFrame = 0;


    void Update()
    {
        if(Input.anyKeyDown)
        {
            //清空按下帧数
            keyFrame=0;
            Debug.Log("任意键被按下");
        }


        if(Input.anyKey)
        {
            keyFrame++;
            Debug.Log("任意键被长按"+keyFrame+"帧");
        }



    }
}
```

在脚本中，当`Input.anyKeyDown`引用的取值为`true`时，表示有任意按键被按下。当`Input.anyKey`引用的取值为`true`时，表示有任意按键处于长按当中。

7.1.5　实例——组合按键

相信大家都玩过经典的拳皇系列游戏，比如经典的连招操作——“下”+“前”+“下”+“前”+“拳”，玩家如果想释放大招技能，就需要在短时间内连续按下一组组合按键。

该需求的实现思路其实很简单：一旦玩家按下某按键后，便开启时间计数器，记录一段时间内每次按下的按键信息（超时表示失败），并在规定时间内对比按键信息与操作集合中的招式，如果满足条件表示释放技能成功，否则表示释放技能失败。如图7-3所示，本例中用户需要根据提示在短时间内按下提示中的方向键，每按下一个按键都会显示在屏幕中，直到按下的按键组合满足下方提示的连续组合按键，才会在屏幕中显示“成功”贴图，超时或按键错误则需要重新按键，具体代码如代码清单7-5所示。



图7-3 组合按键

代码清单7-5 Script_07_05.cs文件

```
using UnityEngine;
using System.Collections.Generic;
using System;

public class Script_07_05 : MonoBehaviour {

    //方向键“上”的贴图
    public Texture imageUp;
    //方向键“下”的贴图
    public Texture imageDown;
    //方向键“左”的贴图
    public Texture imageLeft;
    //方向键“右”的贴图
    public Texture imageRight;
    //按键成功的贴图
    public Texture imageSuccess;

    //自定义方向键的存储值
    public const int KEY_UP = 0;
    public const int KEY_DOWN = 1;
    public const int KEY_LEFT = 2;
    public const int KEY_RIGHT = 3;
    public const int KEY_FIRT = 4;
```

```
//连续按键的事件限制
public const int FRAME_COUNT = 100;
//仓库中存储技能的数量
public const int SAMPLE_SIZE = 3;
//每组技能的按键数量
public const int SAMPLE_COUNT = 5;
//技能仓库
int[,] Sample =
{
    //下 + 前 + 下 + 前 + 拳
    {KEY_DOWN,KEY_RIGHT,KEY_DOWN,KEY_RIGHT,KEY_FIRT},
    //下 + 前 + 下 + 后 + 拳
    {KEY_DOWN,KEY_RIGHT,KEY_DOWN,KEY_LEFT,KEY_FIRT},
    //下 + 后 + 下 + 后 + 拳
    {KEY_DOWN,KEY_LEFT,KEY_DOWN,KEY_LEFT,KEY_FIRT},
};

//记录当前按下按键的键值
int  currentkeyCode =0;
//是否开启监听按键
bool startFrame = false;
//记录当前监听的时间
int  currentFrame = 0;
//保存一段时间内玩家输入的按键组合
List<int> playerSample;
//标志是否完成按键操作
bool isSuccess= false;

void Start()
{
    //初始化按键组合链表
    playerSample  = new List<int>();
}

void OnGUI()
{
    //获得按键组合链表中存储按键的数量
    int size = playerSample.Count;
    //遍历该按键组合链表
    for(int i = 0; i< size; i++)
    {
        //将按下按键对应的图片显示在屏幕中
        int key = playerSample[i];
        Texture temp = null;
        switch(key)
        {
        case KEY_UP:
            temp = imageUp;
            break;
        case KEY_DOWN:
            temp = imageDown;
            break;
        case KEY_LEFT:
            temp = imageLeft;
```

```
                break;
            case KEY_RIGHT:
                temp = imageRight;
                break;
            }
            if(temp != null)
            {
                GUILayout.Label(temp);
            }
        }

        if(isSuccess)
        {
            //显示成功贴图
            GUILayout.Label(imageSuccess);
        }
        //默认提示信息
        GUILayout.Label("连续组合按键1：下、前、下、前、拳");
        GUILayout.Label("连续组合按键2：下、前、下、后、拳");
        GUILayout.Label("连续组合按键2：下、后、下、后、拳");
    }

    void Update()
    {
        //更新按键
        UpdateKey();

        if(Input.anyKeyDown)
        {

            if(isSuccess)
            {
                //按键成功后重置
                isSuccess = false;
                Reset();
            }

            if(!startFrame)
            {
                //启动时间计数器
                startFrame = true;
            }

            //将按键值添加到链表中
            playerSample.Add(currentkeyCode);
            //遍历链表
            int size = playerSample.Count;
            if(size == SAMPLE_COUNT)
            {
                for(int i = 0; i< SAMPLE_SIZE; i++)
                {
                    int SuccessCount = 0;
                    for(int j = 0; j< SAMPLE_COUNT; j++)
                    {
```

7

```
                    int temp = playerSample[j];
                    if(temp== Sample[i,j]){
                     SuccessCount++;
                    }
                }
                //玩家按下的组合按键与仓库中的按键组合相同，表示成功释放技能
                if(SuccessCount ==SAMPLE_COUNT)
                {
                    isSuccess = true;
                    break;
                }
            }

        }
    }

    if(startFrame)
    {
        //递增计数器
        currentFrame++;
    }

    if(currentFrame >= FRAME_COUNT)
    {
        //计数器超时
        if(!isSuccess)
        {
            Reset();
        }
    }
}

void Reset()
 {
    //重置按键相关信息
    currentFrame = 0;
    startFrame = false;
    playerSample.Clear();
 }

void UpdateKey()
{
    //获取当前键盘的按键信息
    if (Input.GetKeyDown (KeyCode.W))
    {
        currentkeyCode = KEY_UP;
    }
    if (Input.GetKeyDown (KeyCode.S))
    {
        currentkeyCode = KEY_DOWN;
    }
    if (Input.GetKeyDown (KeyCode.A))
```

```
            {
                currentkeyCode = KEY_LEFT;
            }
            if (Input.GetKeyDown (KeyCode.D))
            {
                currentkeyCode = KEY_RIGHT;
            }
            if (Input.GetKeyDown (KeyCode.Space))
            {
                currentkeyCode = KEY_FIRT;
            }
        }

    }
```

在上述代码中，我们将用户每次按下的按键信息保存在playerSample链表中，接着在一段时间内判断当前链表中的按键序列是否与正确的按键序列匹配。在Update()方法中，如果时间超过FRAME_COUNT，则表示此次按键时间超时，需要调用Reset()方法清空所有数据，等待用户再次按键。

7.2　鼠标事件

与键盘事件一样，鼠标事件也是PC基本的输入方式。鼠标一般只有3个按键，左键、右键和中键，它操作起来比较灵活，可以任意拖动位置，如在第一人称射击类游戏中，鼠标常用来控制枪的准心，点击左键后可发射子弹。

7.2.1　按下事件

在代码中，可以使用Input.GetMouseButtonDown()来判断鼠标哪个按键被按下，该方法只有一个参数，如果参数为0，则代表鼠标左键被按下，参数为1代表鼠标右键被按下，参数为2代表鼠标中键被按下。此外，使用Input.mousePosition引用可得到鼠标当前位置的三维坐标。如图7-4所示，本例在点击鼠标左键、右键、中键后分别将点击时鼠标的三维坐标打印在屏幕当中，具体代码如代码清单7-6所示。

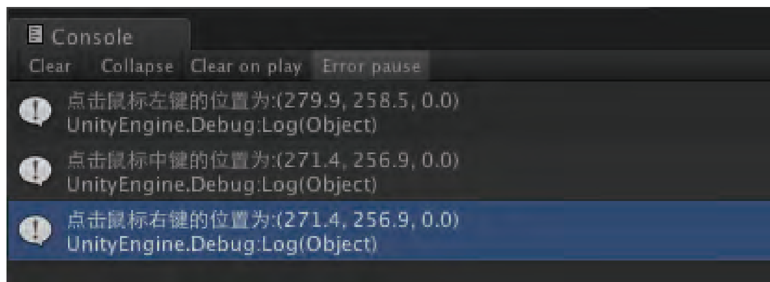


图7-4　按下事件

代码清单7-6　Script_07_06.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_07_06 : MonoBehaviour
{


    void Update()
    {

        if (Input.GetMouseButtonDown(0))
        {
            Debug.Log("点击鼠标左键的位置为:" +Input.mousePosition);
        }
        if (Input.GetMouseButtonDown(1))
        {
            Debug.Log("点击鼠标右键的位置为:" +Input.mousePosition);
        }
        if (Input.GetMouseButtonDown(2))
        {
            Debug.Log("点击鼠标中键的位置为:" +Input.mousePosition);
        }

    }
}
```

7.2.2　抬起事件

鼠标的抬起事件必须依赖按下事件，因为只有在鼠标按下事件后才会出现鼠标抬起事件。在代码中，我们使用`Input.GetMouseButtonUp()`方法监听鼠标按键的抬起事件，其参数和按下事件如出一辙。本例在抬起鼠标左键、右键和中键后将鼠标的三维坐标打印在屏幕中，具体代码如代码清单7-7所示。

代码清单7-7　Script_07_07.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_07_07 : MonoBehaviour
{


    void Update()
    {

        if (Input.GetMouseButtonUp(0))
        {
```

```
                Debug.Log("抬起鼠标左键的位置为:" +Input.mousePosition);
            }
            if (Input.GetMouseButtonUp(1))
            {
                Debug.Log("抬起鼠标右键的位置为:" +Input.mousePosition);
            }
            if (Input.GetMouseButtonUp(2))
            {
                Debug.Log("抬起鼠标中键的位置为:" +Input.mousePosition);
            }

        }
    }
```

7.2.3 长按事件

鼠标长按事件用于监听鼠标三个按键中某按键一直处于按下状态的情况。在代码中，可使用`Input.GetMouseButton()`方法监听鼠标某个按键是否一直处于按下状态。代码清单7-8演示了长按事件的代码。

代码清单7-8 Script_07_08.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_07_08 : MonoBehaviour
{

    //鼠标长按帧数
    int MouseFrame = 0;

    void Update()
    {

        //连按事件
        if(Input.GetMouseButton(0))
        {
            MouseFrame++;
            Debug.Log("鼠标左键长按"+MouseFrame+"帧");
        }
        if(Input.GetMouseButton(1))
        {
            MouseFrame++;
            Debug.Log("鼠标右键长按"+MouseFrame+"帧");
        }
        if(Input.GetMouseButton(2))
        {
            MouseFrame++;
            Debug.Log("鼠标中键长按"+MouseFrame+"帧");
```

```
            }

            //清空长按帧数
            if (Input.GetMouseButtonUp(0))
            {
                MouseFrame = 0;
            }
            if (Input.GetMouseButtonUp(1))
            {
                MouseFrame = 0;
            }
            if (Input.GetMouseButtonUp(2))
            {
                MouseFrame = 0;
            }

        }
    }
```

当鼠标中的某个按键按下时，使用变量将按下的时间帧数记录下来，然后及时显示在屏幕中。为了准确显示按下的时间帧数，在切换按键时将时间帧数清空。

7.3　自定义按键事件

Unity提供了自定义按键的功能，使用时需要在输入管理器中配置自定义按键。与普通按键只能表示一个按键不同，自定义按键以组的形式呈现，并且它可以添加4个不同的按键。按下按键后将返回一个数值，根据这个数值就可以判断按下了自定义按键中的哪个按键。

自定义按键是以“轴”的形式来判断按键事件，比如我们可将“上”、“下”、“左”和“右”4个按钮方向看作两个轴，分别是横向轴与纵向轴。每个轴向可分为正数与负数，按下“上”、“左”轴时返回的值为“1”，按下“下”、“右”轴时返回的值为“–1”，没有按下事件轴时返回的值为“0”，因此可以根据返回的值，很方便地判断自定义按键事件。

7.3.1　输入管理器

输入管理器主要用来配置自定义按键，具体配置方法如下，首先在Unity导航菜单栏中选择“Edit”→“Project Settings”→“Input”菜单项，打开输入管理器界面，然后修改自定义按键的数量（即图7-5中的“Size”项），此时系统会自动添加与删除自定义按键。系统默认提供17个自定义按键，其中大多是常用游戏操作中的一些按键。

本例需要添加一个自定义按键，所以将“Size”的数量修改成18。一般情况下，自定义按键将添加至最后。编辑新添加的自定义按键的名称，暂时将其命名为“test”，然后在下面的矩形框内设置这组自定义按键中包含的4种按键：“a”表示键盘A，“b”表示键盘B，“mouse 0”表示鼠标左键、“Mouse 1”表示鼠标右键。设置完毕后运行游戏，将发现这4种按键都可以影响“test”自定义按键。

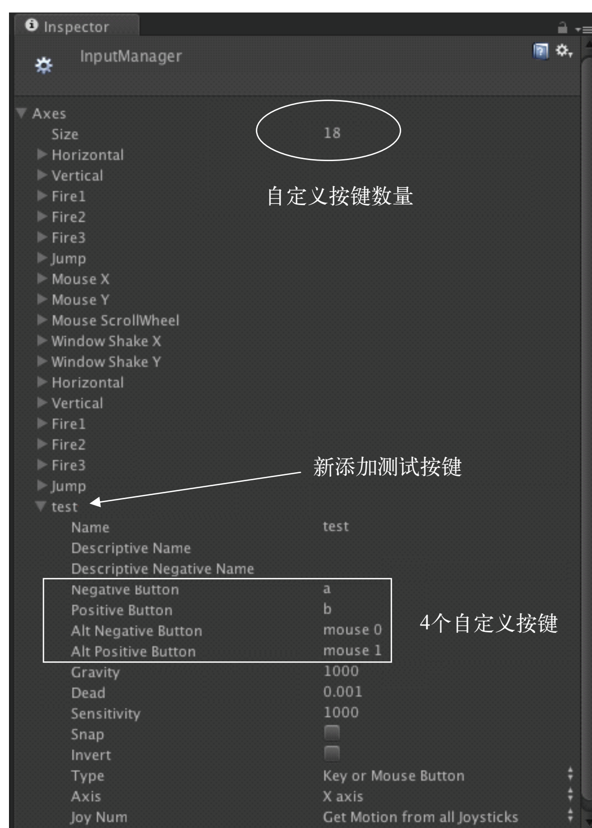


图7-5　输入管理器界面

7.3.2　按键事件

与前面的键盘与鼠标按键事件类似，自定义按键事件使用`Input.GetButtonDown()`方法处理按下事件，使用`Input.GetButton()`方法处理长按事件，使用`Input.GetButtonUp()`方法处理抬起事件。本例首先在输入管理器中创建一个名称为“test”的自定义按键，然后在代码中判断这个自定义按键的所有事件，具体代码如代码清单7-9所示。

代码清单7-9　Script_07_09.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_07_09 : MonoBehaviour
{

    void Update()
    {
```

```
            if (Input.GetButtonDown ("test"))
            {
                Debug.Log("单次按下test自定义按键");
            }

            if (Input.GetButton("test"))
            {
                Debug.Log("长按test自定义按键");
            }

            if (Input.GetButtonUp ("test"))
            {
                Debug.Log("抬起test自定义按键");
            }
        }
    }
```

7.3.3　按键轴

自定义按键可设置轴向，如“上”、“下”、“左”、“右”4个按钮中“上”和“下”确定了一个轴，“左”和“右”确定了一个轴。如图7-6所示，按键轴的数值默认为0，按下某个轴上两边的按钮时，其值分别是“1”与“–1”，根据这个数值可轻易获取玩家按下了哪个方向键，具体代码如代码清单7-10所示。

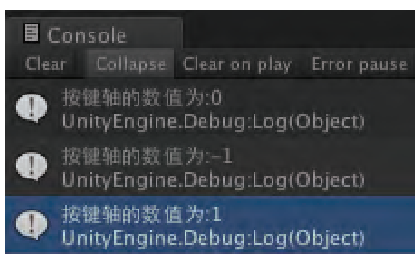


图7-6　按键轴

代码清单7-10　Script_07_10.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_07_10 : MonoBehaviour
{

    void Update()
    {
        float value = Input.GetAxis ("test");
        Debug.Log("按键轴的数值为:"+value);
    }
}
```

在上述代码中，我们使用`Input.GetAxis()`方法获取自定义按键的按下事件，该方法的参数为自定义按键的名称，它应与输入管理器中创建的自定义按钮名称匹配。

7.3.4　实例——观察模型

Unity支持.fbx格式的动画模型，直接将其拖入Hierachy视图即可使用。本例通过时时获取鼠标的位置来旋转主摄像机观察模型，如图7-7所示，旋转鼠标后可在任意角度观察该模型，具体代码如代码清单7-11所示。



图7-7　鼠标旋转

代码清单7-11　Script_07_11.cs文件

7

```
using UnityEngine;
using System.Collections;

public class Script_07_11 : MonoBehaviour
{


    //摄像机参照的模型
    public Transform target;

    //摄像机距离模型的默认距离
    public float distance = 20.0f;

    //鼠标在x轴和y轴方向移动的角度
    float x;
    float y;
```

```
    //限制旋转角度的最小值与最大值
    float yMinLimit = -20.0f;
    float yMaxLimit = 80.0f;

    //鼠标在x和y轴方向移动的速度
    float xSpeed = 250.0f;
    float ySpeed = 120.0f;

    void Start()
    {
        //初始化x和y轴角度，使其等于参照模型的角度
        Vector2 angles = transform.eulerAngles;
        x = angles.y;
        y = angles.x;

        if (rigidbody)
            rigidbody.freezeRotation = true;
    }


    void LateUpdate()
    {
        if (target)
        {
            //根据鼠标移动修改摄像机的角度
            x += Input.GetAxis("Mouse X") * xSpeed * 0.02f;
            y -= Input.GetAxis("Mouse Y") * ySpeed * 0.02f;
            y = ClampAngle(y, yMinLimit, yMaxLimit);
            Quaternion rotation = Quaternion.Euler(y, x, 0);
            Vector3 position = rotation * new Vector3(0.0f, 0.0f, -distance)+
                target.position;
            //设置摄像机的位置与旋转
            transform.rotation = rotation;
            transform.position = position;
            }
    }

    float ClampAngle (float angle , float min , float max)
    {
        if (angle < -360)
            angle += 360;
        if (angle > 360)
            angle -= 360;
        return Mathf.Clamp (angle, min, max);
    }
}
```

在上述代码中，`target`对象为摄像机旋转的参照物，本例中它指向的对象是桥模型。`LateUpdate()`方法用于监听鼠标在屏幕中的坐标，计算出摄像机的位置与旋转的角度，继而实现通过移动鼠标移动摄像机的角度来观察模型。

7.4 模型与动画

模型是3D游戏重要的组成部分，可分为两种：一种是静态模型，如游戏场景中的桌子、椅子等模型；一种是带动画的模型，如主角、敌人等这些会行走、懂AI的模型。模型资源与动画效果需要美工的配合，有关美术方面的知识这里不再赘述。本节主要介绍在Unity中添加模型与播放动画相关的知识。

7.4.1 模型的载入

一般情况下，模型的资源文件由美工提供，其中至少包括模型文件（若此模型存在动画，还需提供动画文件）、材质和贴图。

下面举例介绍一下模型的载入，其中使用的模型源自Unity标准资源包第三方角色控制器组件。为了使读者更清楚载入模型的方式,我将它从包中取出，从头开始载入一遍。如图7-8所示，载入过程为：(1) 将模型的资源与动画文件（Prototype Character）从桌面拖曳到Project视图中；(2) 在Project视图中可以看出该模型由模型文件、材质文件和贴图文件组成，并且已在文件夹中做好分类；(3) 将Project视图中的模型文件拖入Hierarchy视图中；(4) 在Scene视图中即可看到载入的模型文件。运行游戏后，可直接看到模型效果。

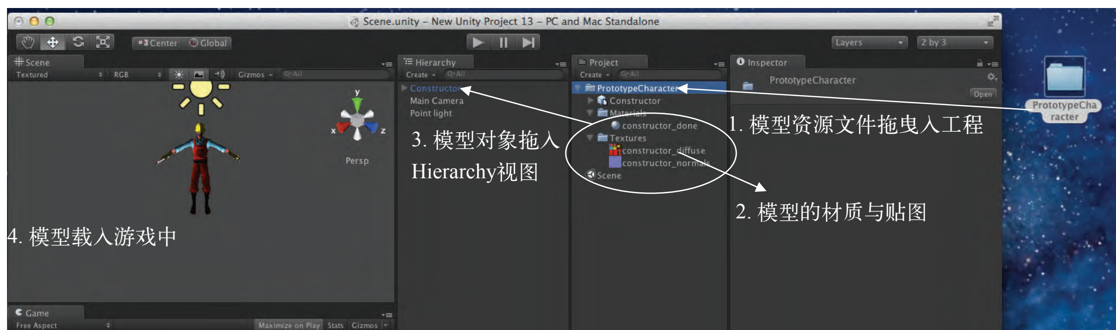


图7-8 载入模型

将模型载入游戏工程后，下面查看该模型资源的动画属性。如图7-9所示，首先在Hierarchy视图中选择“Constructor”模型游戏对象，然后在右侧的Inspector视图中可看到该模型动画的属性，该资源中一共包含4组动画，在“Animation”处可设置模型的默认动画，在“Animations”中可设置该模型包含的其他动画。

在游戏场景中添加模型的方法很简单，只需将Project视图中的模型资源拖曳到Hierarchy视图中，然后在Scene视图中简单调整一下摄像机的位置，缩放一下模型的原始资源，最后在Game视图中将看到这个小人的模型。

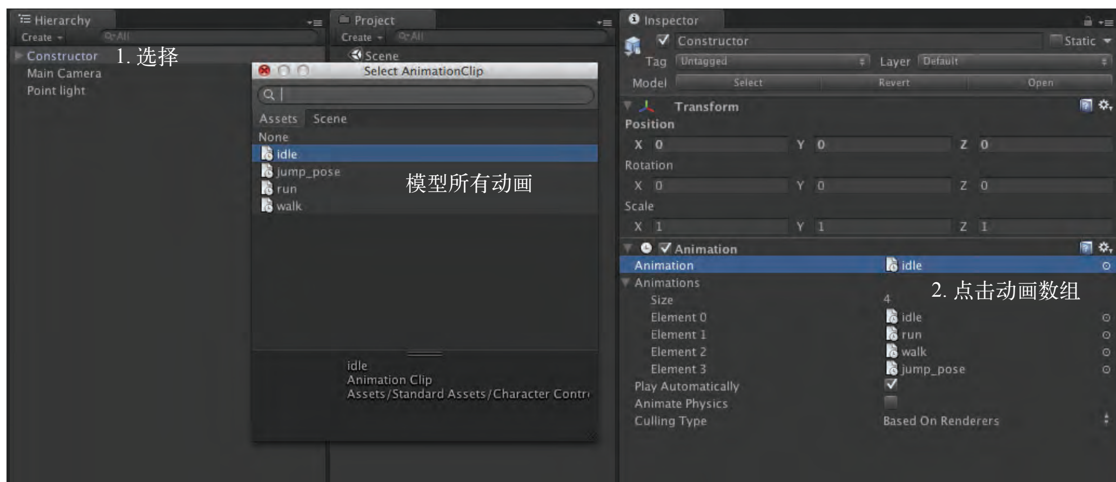


图7-9　查看模型资源的动画属性

7.4.2　设置 3D动画

模型的骨骼动画同样包含在模型中。在Hierarchy视图中选择动画模型后，在右侧的Inspector视图中即可配置模型动画。如图7-10所示，“Animation”（动画）组件中记录着该模型的动画信息，下面简要介绍其中各个属性的含义。

- Animation ：默认的动画名称，如果在播放动画时未指定动画的名称，则播放默认动画。
- Size：动画数量。修改该数值，可添加或删除动画数量，此时下面对应的节点（Element）会随之改变。
- Play Automatically：是否自动播放动画。若勾选该选项，运行游戏后会自动播放默认动画。
- Animate Physics：勾选后表示动画播放时接收物理碰撞。
- Culling Type：模型的类型。

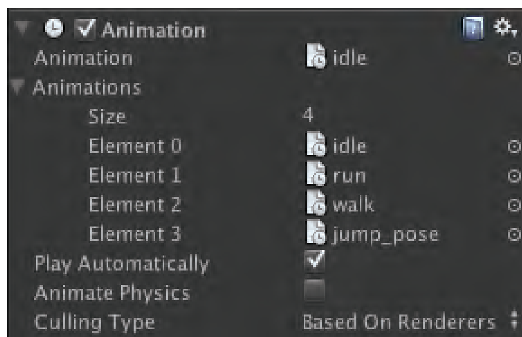


图7-10　配置动画

7.4.3　播放 3D动画

animation.Play()方法用于播放动画，其参数为所播放动画的名称，不写参数表示播放默认动画。如图7-11所示，点击键盘上的字母键“A”、“B”、“C”、“D”，可以切换播放该模型不同的骨骼动画，具体代码如代码清单7-12所示。



图7-11　播放动画

代码清单7-12　Script_07_12.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_07_12 : MonoBehaviour
{
    //动画名称
    public const string ANIM_NAME0="idle";
    public const string ANIM_NAME1="run";
    public const string ANIM_NAME2="walk";
    public const string ANIM_NAME3="jump_pose";
    //模型对象
    private GameObject obj = null;

    void Start()
```

```
    {
        //获取模型动画
        obj = GameObject.Find("Constructor");
        //设置动画播放类型为循环
        obj.animation.wrapMode = WrapMode.Loop;
    }

    void Update()
    {
        //按键后播放不同的动画
        if (Input.GetKeyDown (KeyCode.A))
        {
            obj.animation.Play(ANIM_NAME0);
        }
        if (Input.GetKeyDown (KeyCode.B))
        {
            obj.animation.Play(ANIM_NAME1);
        }
        if (Input.GetKeyDown (KeyCode.C))
        {
            obj.animation.Play(ANIM_NAME2);
        }
        if (Input.GetKeyDown (KeyCode.D))
        {
            obj.animation.Play(ANIM_NAME3);
        }
    }

    void OnGUI()
    {
        //显示提示信息
        GUILayout.Label("点击字母键A、B、C、D切换播放模型骨骼动画");
    }
}
```

在上述代码中，`Update()`方法用于监听按键事件。如果字母键“A”、“B”、“C”、“D”中某一按键被按下，将立即调用`obj.animation.Play()`方法，其中`obj`表示动画模型的游戏对象。`obj.animation.Play()`方法中的参数为需要播放的动画名称，如果动画名称错误，将无法播放动画，并且会抛出异常。

7.4.4　动画剪辑

动画剪辑可以对骨骼动画进行切割播放。实际上，这就是将原始动画剪辑裁剪成一个新剪辑，播放新剪辑就实现了动画的切割播放。此外，动画剪辑还可以合并，但在播放合并剪辑动画时，中间的帧将被去掉。

如图7-12所示，该模型的完整动画为150帧，默认动画为模型完成一次完整的挥杆操作。点击视图中的不同按钮，将播放该动画的不同动画剪辑，具体的代码如代码清单7-13所示。



图7-12 动画剪辑

代码清单7-13 Script_07_13.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_07_13 : MonoBehaviour
{
    //模型对象
    private GameObject obj = null;

    void Start()
    {
        //得到模型动画
        obj = GameObject.Find("man");

    }

    void OnGUI()
    {
        if(GUILayout.Button("播放完整动画"))
        {
            //这里播放默认动画，默认动画即完整的150帧动画
            obj.animation.Play();
        }

        if(GUILayout.Button("切割动画0-50帧"))
        {
            //切割动画，播放第0帧到第50帧
            PlayCuttingAnimation(obj,0,50);
        }
```

```
            if(GUILayout.Button("合并动画0-50帧与100-150帧"))
            {
                //合并动画，将第0帧到第50帧与第100帧到第150帧两组动画合并在一起播放
                PlayCombinedAnimation(obj,0,50,100,150);
            }
        }

        public void PlayCuttingAnimation(GameObject manObject,int startFrame,int
            endFrame)
        {

            AnimationClip clip = manObject.animation.clip;
            //添加一个剪辑，设置起始帧与结束帧
            manObject.animation.AddClip(clip, "cutClip", startFrame, endFrame);
            manObject.animation.Play("cutClip");
        }

        public void PlayCombinedAnimation(GameObject manObject,int startFrame0,int
            EndFrame0,int startFrame1,int EndFrame1)
        {
            AnimationClip clip = manObject.animation.clip;
            //添加两个剪辑，设置起始帧与结束帧

            manObject.animation.AddClip(clip,"startClip",startFrame0,EndFrame0,false);

            manObject.animation.AddClip(clip,"endClip",startFrame1,EndFrame1,false);
            //以队列的形式播放这两个剪辑，保证第一个动画播放完毕后再播放第二个动画
            manObject.animation.PlayQueued("startClip", QueueMode.PlayNow);
            manObject.animation.PlayQueued("endClip", QueueMode.CompleteOthers);
        }
    }
```

在上述代码中，我们首先使用animation.AddClip()方法添加一个动画剪辑，其中该方法的第一个参数表示剪辑的类型，第二个参数表示生成新剪辑的名称，第三个参数表示剪辑的开始帧数，第四个参数表示剪辑的结束帧数，第五个参数表示新剪辑动画是否循环播放，然后以新剪辑的名称作为参数传入，使用animation.Play()播放新的动画剪辑。

如果要在一个动画中截取多个动画剪辑，并且要将它们连续播放，就需要使用一个队列播放动画，这时可以使用animation.PlayQueued()方法，该方法的第一个参数表示播放的动画剪辑的名称，第二个参数表示动画队列的播放模式，其中QueueMode.PlayNow表示立即播放当前剪辑动画，QueueMode.CompleteOthers表示等待其他剪辑动画播放完毕后再播放当前剪辑动画。

7.4.5　动画的帧

任何一组模型动画都是由若干帧组成的。现在已知本例中使用的模型动画的帧数为150帧，那么150帧的播放时间是多久呢？这可以使用方法animation.animation["动画名称"].length来获得，单位为秒。如图7-13所示，在Game视图中左上角处在拖动条中拖动滑块，动画将根据滑块的位置播放特定的动画帧，具体代码如代码清单7-14所示。

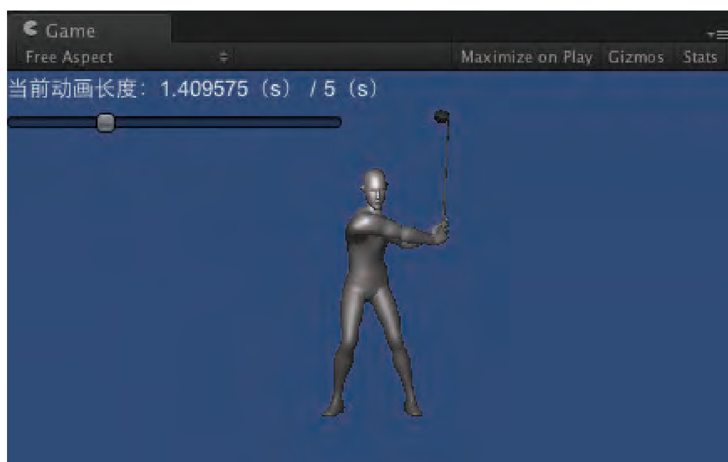


图7-13　动画的帧

代码清单7-14　Script_07_14.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_07_14 : MonoBehaviour
{

    //动画名称
    public const string ANIM_NAME = "Take 001";

    //模型对象
    private GameObject obj = null;

    //拖动条进度
    public float hSliderValue = 0.0f;
    //动画播放长度
    public float animLegth = 0.0f;

    void Start()
    {
        //得到模型动画
        obj = GameObject.Find("man");
        //得到动画播放长度
        animLegth = obj.animation.animation[ANIM_NAME].length;
    }

    void OnGUI()
    {
       //显示信息
        string show = "当前动画长度: "+hSliderValue.ToString() +"(s)"+ " / " +
            animLegth.ToString()+"(s)";
        GUILayout.Label(show);
        //计算拖动条拖动数值
```

```
        hSliderValue = GUILayout.HorizontalSlider(hSliderValue, 0.0F, 5.0F,GUILayout.
            Width(200));
            //绘制动画帧
            PlaySilderAnimation(obj,hSliderValue);

        }

        public void PlaySilderAnimation(GameObject manObject,float times)
        {

            //播放动画
            if(!manObject.animation.IsPlaying(ANIM_NAME))
            {
                manObject.animation.Play(ANIM_NAME);
            }
            //设置动画时间
            manObject.animation.animation[ANIM_NAME].time = times;
        }

    }
```

在上述代码中，我们使用`GUILayout.HorizontalSlider()`方法添加拖动条。拖动滑块后，该方法将返回拖动后的数值。使用`PlaySilderAnimation()`方法播放特定的帧动画，其中该方法的第一个参数表示模型的对象，第二个参数为特定帧的时间。

7.5　GL 图像库

GL图像库是底层的图像库，主要功能是使用程序来绘制常见的2D与3D几何图形。这些图形具有一定的特殊性，它们不属于3D网格图形，只会以面的形式渲染。

使用GL图像库，可在屏幕中绘制2D几何图形，并且该几何图形将永远显示在屏幕当中，不会因为摄像机的移动而改变。2D图形的呈现方式和前面章节介绍的GUI有点类似。值得注意的是，绘制2D图像时，需要使用`GL.LoadOrtho()`方法来将图形映射在平面中；如果绘制的是3D图形，就无须使用此方法。

使用GL图像库时，需要将所有绘制相关的内容写在`OnPostRender()`方法中。此方法由系统自身调用，无法手动调用。此外，有关GL图像库的脚本需要绑定在Hierarchy视图中的摄像机对象当中，否则将无法显示绘制的图形。

7.5.1　绘制线

在了解如何绘制线之前，我们先熟悉一下Unity中GL图像库的平面坐标系。如图7-14所示，按照箭头所指的方向，平面坐标系的原点（0，0）位于左下角。

值得注意的是，GL图像库的平面坐标和普通坐标是有区别的，GL图像库的x轴的最大值是1，y轴的最大值也为1，而不是按照像素来计算的。因此，在GL图像库的平面坐标系中，每个点的横坐标和纵坐标都应当是0与1之间的浮点数，而真实的像素坐标需要根据这个浮点数来计算。

比如当前游戏屏幕的像素宽高是500×500，在GL图像库平面上选择一个点（0.5f，0.5f），那

么这个点真实像素的横坐标和纵坐标应当是：

500（屏幕宽）× 0.5（x坐标）= 250

500（屏幕高）× 0.5（y坐标）= 250

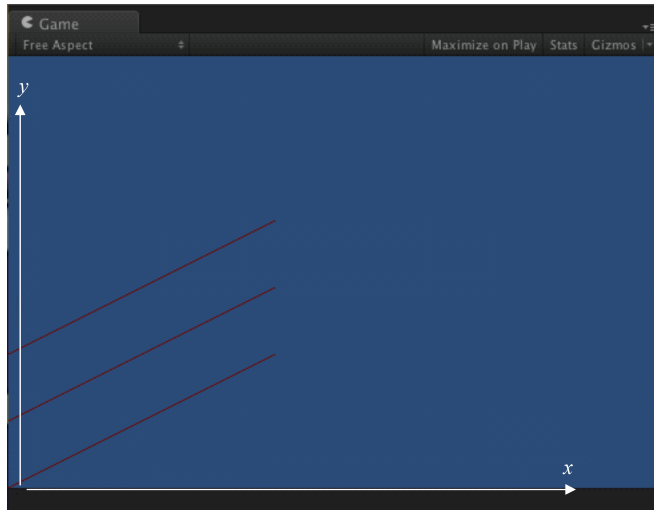


图7-14　绘制线段

本例中，我们在屏幕中绘制了3条朝向为右上角的线段，具体的代码如代码清单7-15所示。

代码清单7-15　Script_07_15.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_07_15 : MonoBehaviour
{

    //绘制线段的材质
    public Material material;

    //此绘制方法由系统调用
    void OnPostRender()
    {
        if (!material)
        {
            Debug.LogError("请给材质资源赋值");
            return;
        }
        //设置该材质通道，0为默认值
        material.SetPass(0);
        //设置绘制2D图像
        GL.LoadOrtho();
        //表示开始绘制，绘制类型为线段
        GL.Begin(GL.LINES);
```

```
            //绘制线段0
            DrawLine(0,0,200,100);
            //绘制线段1
            DrawLine(0,50,200,150);
            //绘制线段2
            DrawLine(0,100,200,200);
            //结束绘制
            GL.End();
        }

        void DrawLine(float x1,float y1,float x2,float y2)
        {
            //绘制线段，需要将屏幕中某个点的像素坐标除以屏幕宽或高
            GL.Vertex(new Vector3(x1/Screen.width, y1/Screen.height, 0));
            GL.Vertex(new Vector3(x2/Screen.width, y2/Screen.height, 0));
        }
    }
```

在上述代码中，我们使用`GL.Begin()`方法开始绘制图形，其中该方法的参数为绘制图形的类型，包括线段、三角形和四边形等。最后需要使用`GL.End()`方法结束图形绘制。此处需要说明的是，所有图形的绘制都必须在这两个方法之间完成。

7.5.2　实例——绘制曲线

本例通过GL图像库记录鼠标移动的轨迹并且将其以曲线的形式显示在屏幕当中，如图7-15所示，具体实现原理是：记录鼠标在Game视图中移动时每一点的坐标，然后将鼠标移动的坐标存储在链表中，使用绘制方法`OnPostRender()`遍历链表中记录的鼠标坐标点，最后通过GL图像库绘制线段的方法将这些点两两连成一条线段即可，具体代码如代码清单7-16所示。

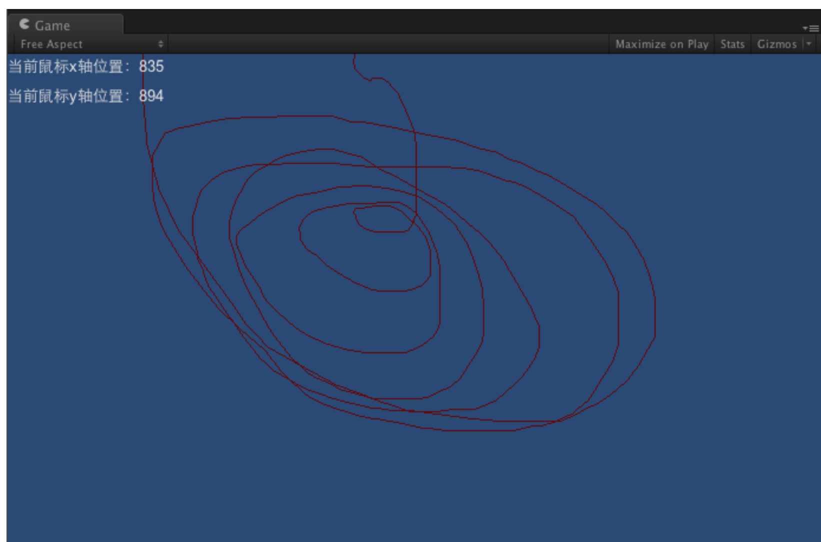


图7-15　绘制曲线

代码清单7-16 Script_07_16.cs文件

```
using UnityEngine;
using System.Collections.Generic;
using System;

public class Script_07_16 : MonoBehaviour
{

    //绘制线段材质
    public Material material;
    private List<Vector3> lineInfo;

    void Start()
    {
        //初始化鼠标线段链表
        lineInfo  = new List<Vector3>();
    }


    void Update()
    {
        //将每次鼠标改变的位置存储进链表
        lineInfo.Add(Input.mousePosition);
    }
    void OnGUI()
    {
         GUILayout.Label("当前鼠标x轴位置："+Input.mousePosition.x);
         GUILayout.Label("当前鼠标y轴位置："+Input.mousePosition.y);
    }

    //此绘制方法由系统调用
    void OnPostRender() {
        if (!material)
        {
            Debug.LogError("请给材质资源赋值");
            return;
        }
        //设置该材质通道，0为默认值
        material.SetPass(0);
        //设置绘制2D图像
        GL.LoadOrtho();
        //表示开始绘制，绘制类型为线段
        GL.Begin(GL.LINES);
        //得到鼠标点信息的总数量
        int size = lineInfo.Count;
        //遍历鼠标点的链表
        for(int i =0; i< size-1; i++)
        {
            Vector3 start = lineInfo[i];
            Vector3 end = lineInfo[i+1];
            //绘制线段
            DrawLine(start.x,start.y,end.x,end.y);
        }
```

```
        //结束绘制
        GL.End();
    }

    void DrawLine(float x1,float y1,float x2,float y2)
    {
        //绘制线段，需要将屏幕中某个点的像素坐标点除以屏幕宽或高
        GL.Vertex(new Vector3(x1/Screen.width, y1/Screen.height, 0));
        GL.Vertex(new Vector3(x2/Screen.width, y2/Screen.height, 0));
    }
}
```

在上述代码中，我们通过`Update()`方法获取当前鼠标的位置，将每帧的鼠标位置存储在`lineInfo`链表中，然后在`OnPostRender()`中遍历这个链表，将链表中记录的鼠标坐标点连接起来绘制在屏幕当中。

7.5.3　绘制四边形

在平面内，由不在同一条直线的四条线段首尾顺序相接组成的图形就是四边形。要确定平面中的一个四边形，需要知道4个点，然后将这4个点连接起来即可。在GL中绘制四边形，需要使用`GL.Begin(GL.QUADS)`方法，该方法中的参数表示需要绘制的图形为四边形。绘制四边形时，需要设置4个合法的点，也就是说这4个点肯定能构成一个四边形。如果设置的4个点在一条直线上，或者只设置了其中3个点，或者两个点重叠，无法让这4个点构成一个四边形，程序就无法绘制该图形，这里需要读者注意一下。

如图7-16所示，本例共绘制了三组几何图形——两个正四边形和一个无规则四边形，具体代码如代码清单7-17所示。

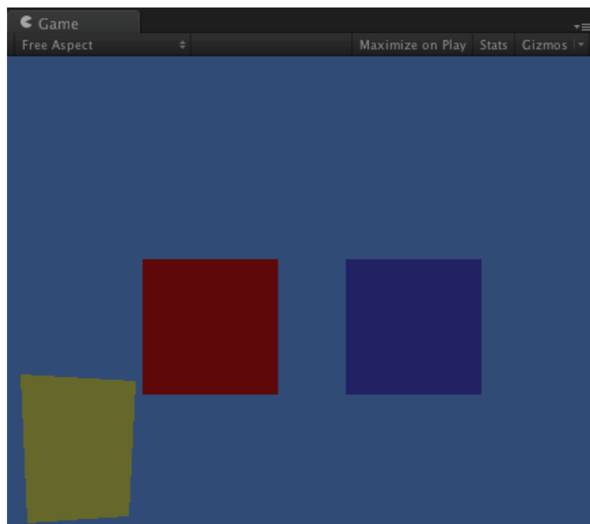


图7-16　绘制的四边形

代码清单7-17 Script_07_17.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_07_17 : MonoBehaviour
{

    //可用材质
    public Material mat0;
    public Material mat1;
    public Material mat3;

    void OnPostRender() {

        //绘制正四边形
        DrawRect(100,100,100,100,mat0);
        DrawRect(250,100,100,100,mat1);
        //绘制无规则四边形
        DrawQuads(15,5,10,115,95,110,90,10,mat3);
    }

    /**
    绘制正四边形
     float x ：x轴起始坐标
     float y ：y轴起始坐标
     float width ：正四边形的宽
     float height ：正四边形的高
    */
    void DrawRect(float x,float y,float width,float height,Material mat)
    {
        GL.PushMatrix();
        mat.SetPass(0);
        GL.LoadOrtho();
        //绘制类型为四边形
        GL.Begin(GL.QUADS);

        GL.Vertex3(x/Screen.width, y/Screen.height, 0);
        GL.Vertex3(x/Screen.width, (y + height)/Screen.height, 0);
        GL.Vertex3((x+ width)/Screen.width, (y + height)/Screen.height, 0);
        GL.Vertex3((x+ width)/Screen.width,y/Screen.height, 0);

        GL.End();
        GL.PopMatrix();
    }

    /**
    绘制无规则的四边形
     float x1 ：起始点1的横坐标
     float y1 ：起始点1的纵坐标
     float x2 ：起始点2的横坐标
     float y2 ：起始点2的纵坐标
     float x3 ：起始点3的横坐标
     float y3 ：起始点3的纵坐标
```

```
     float x4 : 起始点4的横坐标
     float y4 : 起始点4的纵坐标
    */

    void DrawQuads(float x1,float y1,float x2,float y2,float x3,float y3,float x4,
        float y4,Material mat)
    {
        GL.PushMatrix();
        mat.SetPass(0);
        GL.LoadOrtho();
        //绘制类型为四边形
        GL.Begin(GL.QUADS);

        GL.Vertex3(x1/Screen.width, y1/Screen.height, 0);
        GL.Vertex3(x2/Screen.width, y2/Screen.height, 0);
        GL.Vertex3(x3/Screen.width, y3/Screen.height, 0);
        GL.Vertex3(x4/Screen.width, y4/Screen.height, 0);

        GL.End();
        GL.PopMatrix();
    }
}
```

为了说明正四边形和无规则四边形之间的区别，本例将它们封装成两个不同的方法，其中DrawRect()方法用于绘制正四边形，而DrawQuads()方法用于绘制无规则四边形。在上述代码的最后，我们使用GL.End()方法将绘制的四边形显示在屏幕中。

7.5.4　绘制三角形

绘制三角形之前，需要确定平面中的3个点，并且保证这3个点能构成一个三角形，然后将这3个点首尾连接起来即可。绘制三角形时，可以使用GL.Begin(GL.TRIANGLE)方法，该方法的参数为三角形的类型。本例在屏幕中央绘制了一个正三角形，具体代码如代码清单7-18所示。

代码清单7-18　Script_07_18.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_07_18 : MonoBehaviour
{
    //材质
    public Material mat;

    void OnPostRender()
    {
        //绘制三角形
        DrawTriangle(100,0,100,200,200,100,mat);


    }
```

```
    void DrawTriangle(float x1,float y1,float x2,float y2,float x3,float y3,Material
        mat)
    {
        mat.SetPass(0);
        GL.LoadOrtho();
        //绘制三角形
        GL.Begin(GL.TRIANGLES);

        GL.Vertex3(x1/Screen.width, y1/Screen.height, 0);
        GL.Vertex3(x2/Screen.width, y2/Screen.height, 0);
        GL.Vertex3(x3/Screen.width, y3/Screen.height, 0);

        GL.End();
    }

}
```

在上述代码中，我们使用`GL.Vertex3()`方法确定三角形三个顶点的位置，并将绘制三角形的所有方法封装在`DrawTriangle()`方法中，最后使用`GL.End()`方法将三角形显示在屏幕中。需要说明的是，在调用`DrawTriangle()`方法时，需要将三个点的坐标与材质传入该方法。

7.5.5 绘制 3D几何图形

GL图像库不仅支持绘制2D几何图形，还支持绘制3D几何图形，而本例将在3D世界中绘制三个平面四边形。如图7-17所示，为了让读者更方便看出立体效果，我们在Game视图中添加了一个立方体组件作为视图的参照物。通过随时移动鼠标来修改摄像机朝向的位置，可以观察它们之间的区别。圆圈内就是使用GL绘制的图形，它会随着摄像机的位置改变而发生移动，具体的代码如代码清单7-19所示。

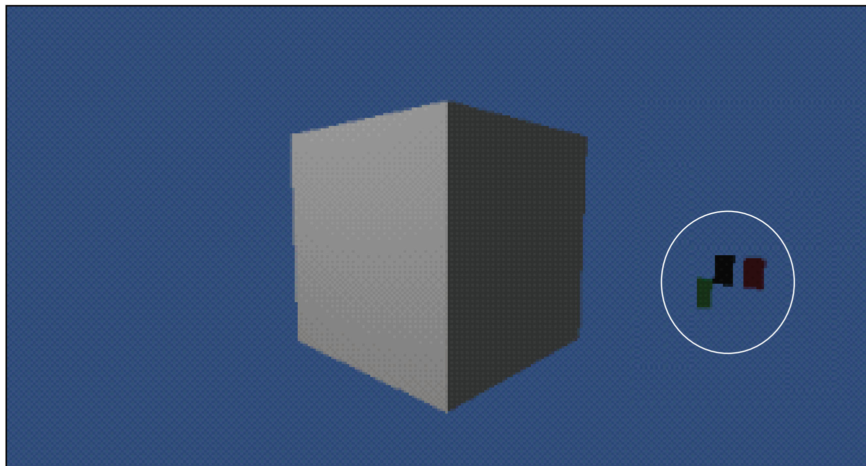


图7-17　立体图形

代码清单7-19　Script_07_19.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_07_19 : MonoBehaviour {

    //可用材质
    public Material mat0;
    public Material mat1;
    public Material mat3;

    void OnPostRender() {

        //绘制正四边形
        DrawRect(100,100,100,100,mat0);
        DrawRect(250,100,100,100,mat1);
        //绘制无规则四边形
        DrawQuads(15,5,10,115,95,110,90,10,mat3);
    }

    /**
    绘制正四边形
     float x ：x轴起始坐标
     float y ：y轴起始坐标
     float width ：正四边形的宽
     float height ：正四边形的高
    */
    void DrawRect(float x,float y,float width,float height,Material mat)
    {
        GL.PushMatrix();
        mat.SetPass(0);
        //绘制类型为四边形
        GL.Begin(GL.QUADS);

        GL.Vertex3(x/Screen.width, y/Screen.height, 0);
        GL.Vertex3(x/Screen.width, (y + height)/Screen.height, 0);
        GL.Vertex3((x+ width)/Screen.width, (y + height)/Screen.height, 0);
        GL.Vertex3((x+ width)/Screen.width,y/Screen.height, 0);

        GL.End();
        GL.PopMatrix();
    }

    /**
    绘制无规则四边形
     float x1 ：起始点1的横坐标
     float y1 ：起始点1的纵坐标
     float x2 ：起始点2的横坐标
     float y2 ：起始点2的纵坐标
     float x3 ：起始点3的横坐标
     float y3 ：起始点3的纵坐标
     float x4 ：起始点4的横坐标
     float y4 ：起始点4的纵坐标
```

```
    */

    void DrawQuads(float x1,float y1,float x2,float y2,float x3,float y3,float x4,
        float y4,Material mat)
    {
        GL.PushMatrix();
        mat.SetPass(0);
        //绘制类型为四边形
        GL.Begin(GL.QUADS);

        GL.Vertex3(x1/Screen.width, y1/Screen.height, 0);
        GL.Vertex3(x2/Screen.width, y2/Screen.height, 0);
        GL.Vertex3(x3/Screen.width, y3/Screen.height, 0);
        GL.Vertex3(x4/Screen.width, y4/Screen.height, 0);

        GL.End();
        GL.PopMatrix();
    }
}
```

在绘制四边形时，首先需要使用`GL.Begin(GL.QUADS)`方法设定渲染模型的类型为四边形，然后使用`GL.Vertex3()`设置四边形每个点的坐标，最后使用`GL.End()`方法将四边形渲染在屏幕当中。

移动摄像机的代码如代码清单7-20所示。

代码清单7-20 MoveCamera.cs文件

```
using UnityEngine;
using System.Collections;


public class MoveCamera : MonoBehaviour
{

    //摄像机参照的模型
    public Transform target;

    //摄像机距离模型的默认距离
    private float distance = 2.0f;

    //鼠标在x轴和y轴方向移动的角度
    float x;
    float y;

    //限制旋转角度的最小值与最大值
    float yMinLimit = -20.0f;
    float yMaxLimit = 80.0f;

    //x和y轴方向的移动速度
    float xSpeed = 250.0f;
    float ySpeed = 120.0f;

    void Start()
```

```
        {
            //初始化x和y轴角度等于参照模型的角度
            Vector2 angles = transform.eulerAngles;
            x = angles.y;
            y = angles.x;

            if (rigidbody)
                rigidbody.freezeRotation = true;
        }


        void LateUpdate()
        {
            if (target)
            {
                //根据鼠标的移动修改摄像机的角度
                x += Input.GetAxis("Mouse X") * xSpeed * 0.02f;
                y -= Input.GetAxis("Mouse Y") * ySpeed * 0.02f;
                y = ClampAngle(y, yMinLimit, yMaxLimit);
                Quaternion rotation = Quaternion.Euler(y, x, 0);
                Vector3 position = rotation * new Vector3(0.0f, 0.0f, (-distance))+ target.
                    position;
                //设置模型的位置与旋转
                transform.rotation = rotation;
                transform.position = position;
            }
        }

        float ClampAngle (float angle , float min , float max)
        {
            if (angle < -360)
                angle += 360;
            if (angle > 360)
                angle -= 360;
            return Mathf.Clamp (angle, min, max);
        }
    }
```

在LateUpdate()方法中通过鼠标的移动来观察模型，该模型的对象保存在target变量当中。

7.5.6　线渲染器

线渲染器主要用于在3D世界中渲染线段。与GL图像库渲染相比，它更加专业，可以控制线段的粗细程度以及线段的数量，并且以网格对象的形式出现在3D世界中。使用线渲染器绘制线段时，必须先确定这条线段两个端点的位置。需要说明的是，这两个点不是平面中的点而是3D世界中的点。

线渲染器以组件的形式出现在Unity当中，所以需要将它绑定在某个游戏对象中。这里我们在Unity导航菜单栏中选择“GameObject”→“CreateEmpty”菜单项创建一个空的游戏对象，然

后在Hierarchy视图中选择该对象后，继续在Unity导航菜单栏中选择“Component”→“Miscellaneous”→“Line Renderer”菜单项，即可将线渲染器组件添加至该游戏对象中，如图7-18所示。

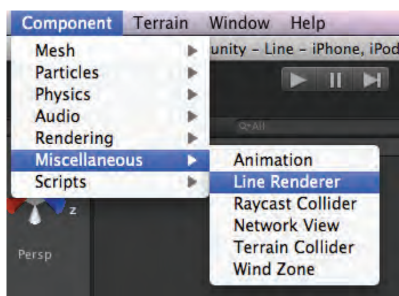


图7-18 添加线渲染器组件

然后选择该游戏对象，在右侧的Inspector视图中可以看到线渲染器组件的参数，如图7-19所示。下面我们先来简要介绍一下这些参数的含义。

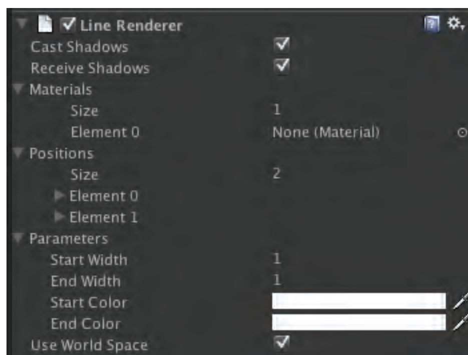


图7-19 添加线渲染器

- Cast Shadows：是否投射阴影。
- Receive Shadows：是否接收阴影。
- Materials：设置材质，这里可以设置多个材质并且它们依次排开。
- Positions：这个属性就比较重要了，它专门用于设置线段在3D 世界中的坐标，其中Size与线段的数量保持一致，Element节点中就是每个线段点的位置。
- Start Width：设置线段起点的宽度。
- End Width：设置线段终点的宽度。
- Start Color：设置起点颜色。
- End Color：设置终点颜色。
- Use World Space：使用世界坐标系。

本例中我们绘制了3条相连的线段，它是以4个顶点确定的3条线段，并且它们首尾相接成一条直线状。这个线段以立体的形式出现在3D世界当中，如图7-20所示，具体代码如代码清单7-21所示。

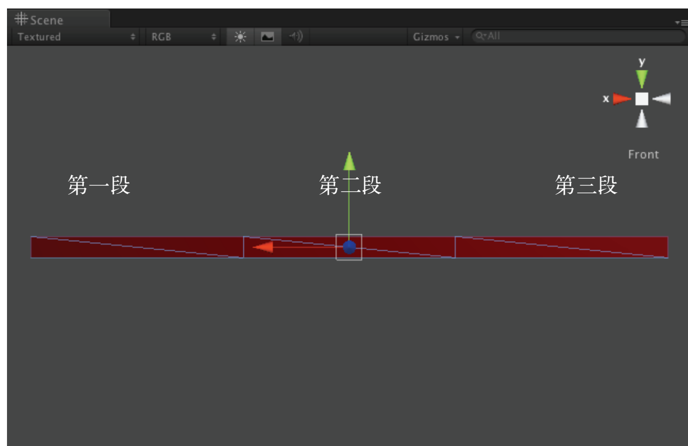


图7-20　绘制线段

代码清单7-21　Script_07_20.cs文件

```
using UnityEngine;
using System.Collections;
using System.Threading;

public class Script_07_20 : MonoBehaviour
{
    //线段对象
    private GameObject LineRenderGameObject;

    //线段渲染器
    private LineRenderer lineRenderer;

    //设置线段的顶点数，4个点确定3条线段
    private int lineLength = 4;

    //记录4个点，连接一条线段
    private Vector3 v0 = new Vector3(1.0f,0.0f,0.0f);
    private Vector3 v1 = new Vector3(2.0f,0.0f,0.0f);
    private Vector3 v2 = new Vector3(3.0f,0.0f,0.0f);
    private Vector3 v3 = new Vector3(4.0f,0.0f,0.0f);

    void Start()
    {
        //获得线段游戏对象
        LineRenderGameObject = GameObject.Find ("ObjLine");
        //获得线渲染器组件
        lineRenderer = (LineRenderer)LineRenderGameObject.GetComponent ("LineRenderer");
```

```
        //设置线的顶点数
        lineRenderer.SetVertexCount(lineLength);
        //设置线的宽度
        lineRenderer.SetWidth(0.1f,0.1f);
    }

    void Update() {

        //使用这4个顶点渲染3条线段
        lineRenderer.SetPosition (0, v0);
        lineRenderer.SetPosition (1, v1);
        lineRenderer.SetPosition (2, v2);
        lineRenderer.SetPosition (3, v3);

    }

}
```

在上述代码中，我们首先获取了线渲染器组件对象，然后设置顶点的数量，最后调用`SetPosition()`方法将线段显示在屏幕当中。`SetPosition()`方法的第一个参数表示每个点的ID，让它保持唯一性，第二个参数表示该顶点的3D位置。

7.5.7 网格渲染

任何一个模型都由若干网格面组成，而每一个面又由若干个三角形组成，也就是说，模型是由若干个三角形面组成的。本节我们将学习如何使用三角形创建一个网格面。

创建网格面的方式如下：首先选择一个游戏对象，为其添加Mesh（网格）属性与Mesh Filter（网格过滤器）属性。这里我们在Unity导航菜单栏中选择“GameObject”→“CreateEmpty”菜单项创建一个空的游戏对象，然后在Hierarchy视图中选择该对象，接着在Unity导航菜单栏中选择“Component”→“Mesh”→“Mesh Filter”菜单项与“Mesh Renderer”菜单项，即可将网格渲染组件添加至该游戏对象本身，如图7-21所示。

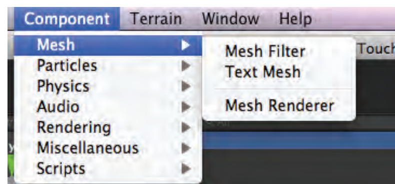


图7-21 绘制线段

如图7-22所示，本例在屏幕中渲染了两个网格面对象。因为网格面由三角形组成，所以需要得知三角形三个顶点的位置。代码中`mesh.vertices`数组用于保存三角形网格顶点的位置，三角形由3个顶点组成，所以它们的规律是：一个三角形数组长度就是3，两个三角形数组长度就是6，依次类推该数组的长度只可能是3的倍数。最后绘制网格时使用triangles数组，数组中的ID和Vertices数组中的顶点ID一一对应。



图7-22　绘制网格

本例代码如代码清单7-22所示。

代码清单7-22　Script_07_21.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_07_21 : MonoBehaviour
{


    //构成三角形1的三个顶点位置
    Vector3 v0 = new Vector3(5, 0, 0);
    Vector3 v1 = new Vector3(0, 5, 0);
    Vector3 v2 = new Vector3(0, 0, 5);

    //构成三角形2的三个顶点位置
    Vector3 v3 = new Vector3(-5, 0, 0);
    Vector3 v4 = new Vector3(0, -5, 0);
    Vector3 v5 = new Vector3(0, 0, -5);


    //构成三角形1的贴图比例
    Vector2 u0 = new Vector2(0, 0);
    Vector2 u1 = new Vector2(0, 5);
    Vector2 u2 = new Vector2(5, 5);

     //构成三角形2的贴图比例
    Vector2 u3 = new Vector2(0, 0);
    Vector2 u4 = new Vector2(0, 1);
    Vector2 u5 = new Vector2(1, 1);

    void Start()
    {

        //得到网格渲染器对象
```

```
            MeshFilter meshFilter = (MeshFilter)GameObject.Find("face").GetComponent
                (typeof(MeshFilter));

            //通过渲染器对象得到网格对象
            Mesh mesh = meshFilter.mesh;

            //设置三角形顶点的数组，6个点表示设置了两个三角形
            mesh.vertices = new Vector3[] {v0, v1, v2,v3, v4, v5};

            //设置三角形面上的贴图比例
            mesh.uv = new Vector2[] {u0, u1, u2,u3, u4, u5};

            //设置三角形索引，绘制三角形
            mesh.triangles= new int [](0,1,2,3,4,5};

        }
    }
```

代码最后的mesh.triangles表示设定三角形的索引数组，该数组中的ID表示相对顶点数组中的坐标。目前这个数组中的元素是0、1、2、3、4和5，对应顶点数组中6个顶点坐标。因为3个点确定一个三角形面，所以这里使用定点数组中0、1、2确定了一个三角形，3、4、5又确定了一个三角形。

7.6 游戏实例——控制人物移动

为了让读者更清晰地了解如何控制主角移动与播放骨骼动画，下面我们将角色控制器组件中的人物动画拆开，使用代码自行实现他的行走动画。运行游戏后，按下键盘键上的“W”、“S”、“A”和“D”来移动主角，如图7-23所示。本例代码如代码清单7-23所示。

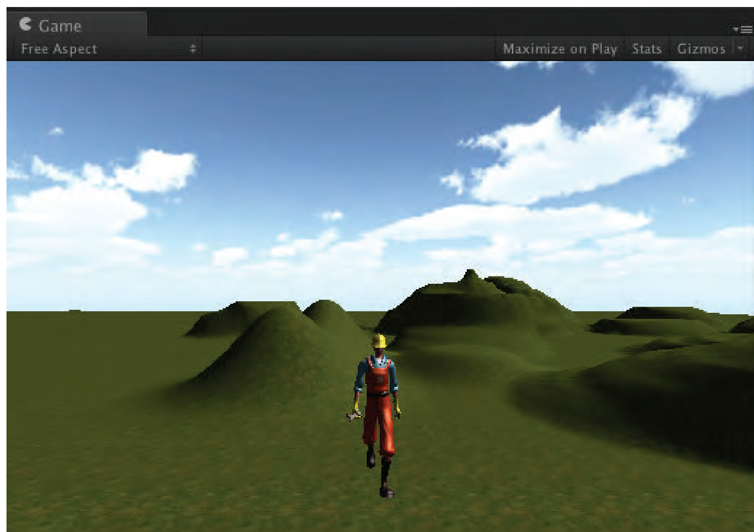


图7-23 实例效果

代码清单7-23　Script_07_22.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_07_22 : MonoBehaviour {

    //人物的行走方向状态
    public const int HERO_UP= 0;
    public const int HERO_RIGHT= 1;
    public const int HERO_DOWN= 2;
    public const int HERO_LEFT= 3;

    //人物当前的行走方向状态
    public int state = 0;

    //人物移动速度
    public int moveSpeed = 10;

    //初始化人物的默认位置
    public void Awake()
    {
    state = HERO_DOWN;
    }

    void Update() {

        //获取控制的方向数据
        float KeyVertical =  Input.GetAxis ("Vertical") ;
        float KeyHorizontal =  Input.GetAxis ("Horizontal");

        if(KeyVertical == -1)
        {
            //设置人物动画为向左行走
            setHeroState(HERO_LEFT);

        }else if(KeyVertical == 1)
        {
            //设置人物动画为向右行走
            setHeroState(HERO_RIGHT);
        }

        if(KeyHorizontal == 1)
        {
            //设置人物动画为向后行走
            setHeroState(HERO_DOWN);

        }else if(KeyHorizontal == -1)
        {
            //设置人物动画为向前行走
            setHeroState(HERO_UP);
        }

        if(KeyVertical == 0 && KeyHorizontal ==0)
        {
            //松开按键播放默认动画
```

```
                animation.Play();
            }

        }

        public void setHeroState(int newState)
        {

            //根据当前人物方向与上一次备份方向计算出模型旋转的角度
            int rotateValue = (newState - state) * 90;
            Vector3 transformValue = new Vector3();

            //播放行走动画
            animation.Play("walk");

            //模型移动的位移的数值
            switch(newState){
                case HERO_UP:
                transformValue = Vector3.forward * Time.deltaTime;
                break;
                case HERO_DOWN:
                transformValue = (-Vector3.forward) * Time.deltaTime;
                break;
                case HERO_LEFT:
                transformValue = Vector3.left * Time.deltaTime;
                break;
                case HERO_RIGHT:
                transformValue = (-Vector3.left) * Time.deltaTime;
                break;
            }

            //模型旋转
            transform.Rotate(Vector3.up, rotateValue);
            //移动人物
            transform.Translate(transformValue * moveSpeed, Space.World);

            state = newState;

        }

    }
```

本例使用游戏状态机将主角的行走分为4个状态：向前行走、向后行走、向左行走、向右行走。按下不同的方向键后，使用`animation.Play()`方法播放行走动画，该方法的参数为动画的名称，最后根据当前的行走状态计算模型的旋转角度，使其按照正确的方向行走。

7.7 本章小结

本章首先介绍了如何处理键盘与鼠标输入事件，比如按下事件、抬起事件和长按事件等，接着介绍了自定义按键事件、模型与动画，然后介绍了如何使用GL图像库绘制2D与3D的线段与网格模型，以及线渲染器与网格渲染器的绘制方法，最后以一个实例的形式向读者介绍如何使用键盘控制主角移动并且播放骨骼动画。

第8章 持久化数据

喜欢玩游戏的朋友应该很清楚，很多游戏中都会出现与“存储进度”、“读取进度”相关的菜单项，玩家一般将其称为游戏存档功能。它们用于保存玩家的游戏数据，并且还可以通过读取之前的游戏进度继续游戏，这就牵扯到本章的核心内容——数据存储。一般情况下，系统只存储一些轻量级的数据，比如字符型、整型、浮点型数据等。这是因为数据类型越简单，存储与读取就越迅速。此外，所有数据的存储都是通过流的形式来完成的，系统使用流将数据写入文件或从文件读取至程序。

8.1 PlayerPrefs 类

Unity提供了一个用于本地持久化保存与读取数据的类——PlayerPrefs。它的工作原理是以键值对的形式将数据保存在文件中，这就好比给需要保存的每一个数据赋予名称，在将其成功存入本地存档后，程序就可以根据这个名称取出上次保存的数据。

8.1.1 保存与读取数据

PlayerPrefs类可保存与读取3种基本的数据类型，它们是浮点型、整型和字符串型，涉及的方法如下。

- SetFloat()：保存浮点类型。
- SetInt()：保存整型。
- SetString()：保存字符串。
- GetFloat()：获取浮点类型。
- GetInt()：获取整型。
- GetString()：获取字符串。

这些函数的用法基本相同，使用Set进行保存，使用Get进行读取。下面列举一个存储与读取整型数据的例子：

```
//保存整型
PlayerPrefs.SetInt("Test", 100);

//读取数据
int i = PlayerPrefs.GetInt("Test",888);
```

保存数据时，需要调用对应的Set方法，该方法的第一个参数表示存储数据的名称，第二个参数表示具体存储的数值；读取数据时，需要调用对应的Get方法，其返回值就是读取后的数值，该方法的第一个参数表示读取数据的名称，第二个参数表示读取数据的默认值，即如果通过数据名称没有找到对应的值，那么就直接返回数据的默认值。

8.1.2　删除数据

此外，PlayerPrefs类还提供了两种删除数据的方法：第一种为删除某一项数据，第二种为删除所有数据。为了安全起见，删除前还需判断系统中是否存有该数据。删除数据的方法如下。

- DeleteAll()：将所有键对应的值数据统统删除。
- DeleteKey()：删除某一项数据，其参数为所删除数据的名称。
- HasKey()：判断是否存有这项数据，其参数为判断的数据名称。

8.1.3　实例——注册界面

本节将通过一个实例向读者充分诠释数据的存储与读取，如图8-1所示，在相关注册信息后输入完成后，用户可通过点击“提交数据”按钮将数据持久化保存，所保存的信息将显示在屏幕下方，点击“取消查看”按钮，即可关闭显示信息，并且清空所有持久化存储信息，具体代码如代码清单8-1所示。

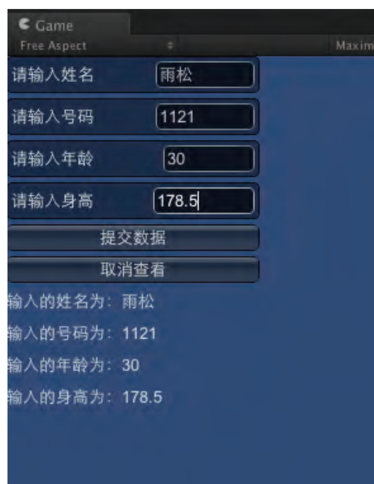


图8-1　存储与读取数据

代码清单8-1　Script_08_01.cs文件

```
using UnityEngine;
using System.Collections;
```

```
public class Script_08_01 : MonoBehaviour
{

    //用户姓名
    private string username="";
    //用户号码
    private string usernumber="";
    //用户年龄
    private string userage="";
    //用户身高
    private string userheight="";
    //是否将信息显示
    private bool showInfo = false;

    void OnGUI()
    {
        GUILayout.BeginHorizontal ("box",GUILayout.Width(200));

        GUILayout.Label ("请输入姓名");
        username = GUILayout.TextField(username, 10);
        GUILayout.EndHorizontal();

        GUILayout.BeginHorizontal ("box");
        GUILayout.Label ("请输入号码");
        usernumber = GUILayout.TextField(usernumber, 11);
        GUILayout.EndHorizontal();

        GUILayout.BeginHorizontal ("box");
        GUILayout.Label ("请输入年龄");
        userage = GUILayout.TextField(userage, 2);
        GUILayout.EndHorizontal();

        GUILayout.BeginHorizontal ("box");
        GUILayout.Label ("请输入身高");
        userheight = GUILayout.TextField(userheight, 5);
        GUILayout.EndHorizontal();

        if(GUILayout.Button ("提交数据"))
        {
            showInfo = true;
            //持久化保存数据
            PlayerPrefs.SetString("username",username);
            PlayerPrefs.SetString("usernumber",usernumber);
            PlayerPrefs.SetInt("userage",int.Parse(userage));
            PlayerPrefs.SetFloat("userheight",float.Parse(userheight));
        }

        if(GUILayout.Button ("取消查看"))
        {
            showInfo = false;
            //删除所有持久化对应值
            PlayerPrefs.DeleteAll();
        }
```

```
        if(showInfo)
        {
            //将信息显示出来，如果找不到对应的值则显示信息默认值
            GUILayout.Label ("输入的姓名为：" + PlayerPrefs.GetString("username","
                姓名默认值"));
            GUILayout.Label ("输入的号码为：" + PlayerPrefs.GetString("usernumber","
                号码默认值"));
            GUILayout.Label ("输入的年龄为：" +
                PlayerPrefs.GetInt("userage",0).ToString());
            GUILayout.Label ("输入的身高为：" +
                PlayerPrefs.GetFloat("userheight",0.0f).ToString());
        }

    }
}
```

本例中，在用户点击“提交数据”按钮后，程序开始读取输入框中的内容，因为输入框的数据类型为字符串型，为了将其保存为整型或者浮点型数据，系统需要进行强制转换，以上代码使用`int.Parse(string)`方法将字符串强转为整型，使用`float.Parse(string)`方法将字符串强转为浮点型。数据类型转换完成后，使用`PlayerPrefs`类保存所有数据即可。点击“取消查看”按钮将删除所有保存的数据，在代码中可以使用`PlayerPrefs.DeleteAll()`方法来实现。

8.2　自定义文件

如果需要在程序中处理大量的数据，使用`PlayerPrefs`类就不太合适了，因为`PlayerPrefs`是轻量级的存储。在存储电子小说这类文字资源非常海量的数据时，可以先将文字数据保存在本地文件中，再通过流去读取本地的资源数据，然后将其显示在屏幕中。此外，使用流还可以在程序中动态地创建本地文本文件，也可以将字符串信息动态地写入本地文本文件。本节将主要向读者介绍自定义文件的相关处理方法。

8.2.1　文件的创建与写入

文件的创建与写入都需要使用流来操作。要创建与写入文件，首先需要3组相关信息：创建文件的路径、名称以及写入文件的内容。本例将使用流来创建一个文本文件，并且将3组数据写入其中，具体代码如代码清单8-2所示。

代码清单8-2　Script_08_02.cs文件

```
using UnityEngine;
using System.Collections;
using System.IO;

public class Script_08_02 : MonoBehaviour
{

    void Start()
    {
```

```
        //创建文件，共写入3次数据
        CreateFile(Application.dataPath,"FileName","TestInfo0");
        CreateFile(Application.dataPath,"FileName","TestInfo1");
        CreateFile(Application.dataPath,"FileName","TestInfo2");
     }

    /**
    * path：文件创建目录
    * name：文件的名称
    * info：写入的内容
    */
    void CreateFile(string path,string name,string info)
    {
       //文件流信息
       StreamWriter sw;
       FileInfo t = new FileInfo(path+"//"+ name);
       if(!t.Exists)
       {
         //如果此文件不存在则创建
         sw = t.CreateText();
       }
       else
       {
         //如果此文件存在，则打开该文件
         sw = t.AppendText();
       }
       //以行的形式写入信息
       sw.WriteLine(info);
       //关闭流
       sw.Close();
       //销毁流
       sw.Dispose();
    }

}
```

创建本地文件时，需要使用`FileInfo`类，在构造方法中写入文件的保存路径。通过对`FileInfo`的对象使用`CreateText()`方法，可在本地创建一个文本文件，使用`AppendText()`方法可打开已创建的文本文件，最后调用`WriteLine()`方法可将字符串写入刚刚创建的文本文件中。

8.2.2　文件的读取

文件的读取和写入一样，需要使用流来处理。文本文件中的数据都是逐行存储的，因此可以使用流将文本内容逐行读出。本例使用程序按行读取本地文本文件中的数据，最后逐行输出显示，具体代码如代码清单8-3所示。

代码清单8-3　Script_08_03.cs文件

```
using UnityEngine;
using System.Collections;
```

```
using System.Collections.Generic;
using System.IO;
using System;

public class Script_08_03 : MonoBehaviour {

    void Start()
    {
        //读取文件
        ArrayList info = LoadFile(Application.dataPath,"FileName");

        //遍历文本信息，将其打印出来
        foreach(string str in info)
        {
            Debug.Log(str);
        }
    }

    /**
    * path：读取文件的路径
    * name：读取文件的名称
    */
    ArrayList LoadFile(string path,string name)
    {
        //使用流读取
        StreamReader sr =null;
        try{
            sr = File.OpenText(path+"//"+ name);
        }catch(Exception e)
        {
            //通过路径与名称均未找到文件，则直接返回空
            return null;
        }
        string line;
        ArrayList arrlist = new ArrayList();
        while ((line = sr.ReadLine()) != null)
        {
            //逐行读取
            //将每一行的内容存入数组链表容器中
            arrlist.Add(line);
        }
        //关闭流
        sr.Close();
        //销毁流
        sr.Dispose();
        //返回数组链表容器
        return arrlist;
    }
}
```

读取文件时，首选需要获取文件流，然后以循环的方式通过`sr.ReadLine()`方法将文本文件中的内容全部按行读取出来，读取完毕后该方法将返回`null`。

8.2.3　实例——读取笑话

本节将通过一个实例向读者介绍文件读取的具体步骤。首先在本地保存了5个文本文件，其中包含一些笑话。值得注意的是，这里需要将文本内容保存为UTF-16，否则会出现乱码。

在文本文件中不同的笑话之间使用“&”做分隔符号，在程序中使用流将这些文本笑话读取并且分割显示在屏幕当中。如图8-2所示，最上方的5个按钮用来读取不同的笑话文本文件，点击“显示内容/取消”按钮将读取出笑话的内容，使用GUI将内容显示在屏幕当中，本例代码如代码清单8-4所示。

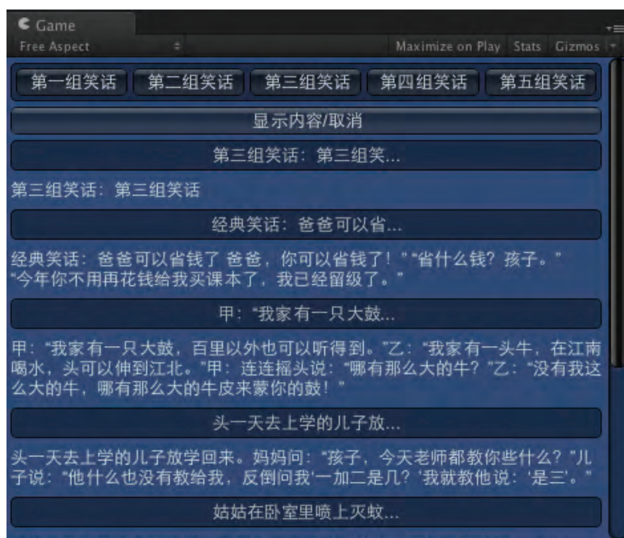


图8-2　读取笑话

代码清单8-4　Script_08_04.cs文件

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using System.IO;
using System;

public class Script_08_04 : MonoBehaviour {

    //所有笑话都存储在该链表中
    ArrayList info = null;
    //是否展示所有信息
    bool button = false;
    //滚动视图拖动条
    public Vector2 scrollPosition;

    void Start()
    {
```

```
        //读取文件
        info = LoadFile(Application.dataPath,"FileName0");
        //初始化滚动拖动条
        scrollPosition = new Vector2(0.0f,0.0f);

    }

    /**
    * path：读取文件的路径
    * name：读取文件的名称
    */
    ArrayList LoadFile(string path,string name)
    {
        //使用流读取
        StreamReader sr =null;
        try{
            sr = File.OpenText(path+"//"+ name);
        }catch(Exception e)
        {
            //通过路径与名称均未找到文件，则直接返回空
            e.ToString();
            return null;
        }
        string line;
        ArrayList arrlist = new ArrayList();
        //逐行读取笑话
        while ((line = sr.ReadLine()) != null)
        {
            //使用"&"将笑话分割
            string [] str = line.Split(new char[]{'&'});

            foreach (string i in str)
            {
                //将每段笑话添加到链表中
                arrlist.Add(i);
            }

        }
        //关闭流
        sr.Close();
        //销毁流
        sr.Dispose();
        //返回数组链表容器
        return arrlist;
    }

    void OnGUI()
    {
        //开始滚动视图
        scrollPosition = GUILayout.BeginScrollView(scrollPosition, GUILayout.Width
            (Screen.width), GUILayout.Height(Screen.height));
        GUILayout.BeginHorizontal ("box");
        if(GUILayout.Button("第一组笑话"))
        {
        //读取文件
        info = LoadFile(Application.dataPath,"FileName0");
        }
```

```
            if(GUILayout.Button("第二组笑话"))
            {
                //读取文件
                info = LoadFile(Application.dataPath,"FileName1");
            }
            if(GUILayout.Button("第三组笑话"))
            {
                //读取文件
                info = LoadFile(Application.dataPath,"FileName2");
            }
            if(GUILayout.Button("第四组笑话"))
            {
                //读取文件
                info = LoadFile(Application.dataPath,"FileName3");
            }
            if(GUILayout.Button("第五组笑话"))
            {
                //读取文件
                info = LoadFile(Application.dataPath,"FileName4");
            }
            GUILayout.EndHorizontal();

            if(GUILayout.Button("显示内容/取消"))
            {
                if(button)
                    button = false;
                else
                    button = true;
            }

            //遍历所有笑话
            int size = info.Count;
            for(int i=0;i < size; i++)
            {
                //得到每个笑话的所有内容
                string text = (string)info[i];
                //因为每个笑话的内容过多，所以截取笑话
                string title = text.Substring(0,10)+"...";
                //显示笑话标题
                GUILayout.Box(title);

                if(button)
                {
                    //显示笑话的全部内容
                    GUILayout.Label(text);
                }
            }
            //结束滚动视图
            GUILayout.EndScrollView ();
        }

    }
```

为了避免文本内容过多而超出屏幕，本例将所有文本内容写入滚动视图当中，然后使用循环LoadFile()方法将本地文件中的所有笑话数据存入数组链表中，最后再使用OnGUI()方法将所有笑话数据绘制在屏幕当中。

8.3 应用程序

应用程序是一项非常重要的属性，所有应用程序相关的方法都写在`Application`类中。它可以获取或设置当前程序的一些属性，比如加载游戏关卡，获取资源文件路径，退出当前游戏程序，获取当前游戏平台等。

应用程序的状态可分为3种，它们分别是：程序获取焦点时，表示当前游戏处于可控制状态；程序暂停时，表示当前游戏呈不可控制状态；程序退出时，表示当前游戏已经退出。在脚本中可以使用下列方法监听这3种应用程序的状态。

- `OnApplicationFocus()`：获得焦点时。
- `OnApplicationPause()`：程序暂停时。
- `OnApplicationQuit()`：程序退出时。

8.3.1 创建关卡

游戏关卡也可以称为游戏场景。任何一个完整的游戏都是由若干场景组成的，比如游戏菜单场景、游戏主界面场景、战斗场景、胜利场景和通关场景等。通过这些场景的组合，可将整个游戏拆分成若干个模块，而在程序中可使用代码直接切换这些模块。

使用Unity创建一个新游戏项目，然后在导航菜单栏中选择“File”→“New Scene”菜单项创建一个新关卡（同一个游戏项目中可以存在任意数量的游戏关卡），然后系统默认会将主摄像机加入到场景中，随后可以在该场景中任意编写脚本或编辑地图，最后使用快捷键“Command＋S”保存在对应场景中即可。

8.3.2 切换关卡

如需使用代码切换关卡，首先需要在应用程序中授权该关卡。在Unity导航菜单栏中选择“File”→“Build Settings”菜单项，在打开的界面中点击右下角的“Add Current”按钮可为当前游戏关卡授权，上面方框内的内容就是授权过的关卡信息，右侧数字为关卡的ID，如图8-3所示。

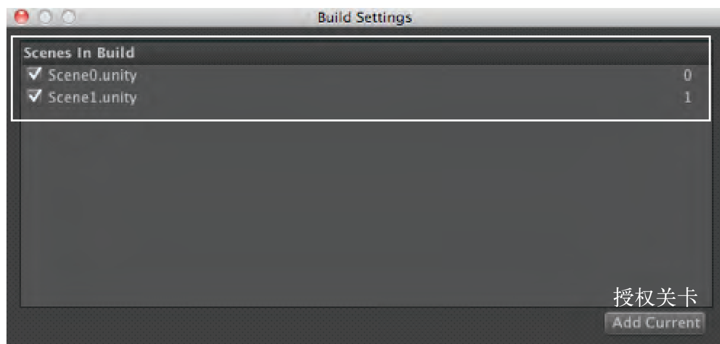


图8-3 授权关卡

值得注意的是，ID为0的关卡表示程序运行时第一个进入的场景。选择某个关卡后，使用鼠标上下拖曳即可修改其ID。选中某个关卡，按下键盘上的“delete”键即可删除该关卡。

本工程共创建了两个游戏关卡，点击下方的按钮，可切换当前关卡，如图8-4所示。本例的代码如代码清单8-5所示。

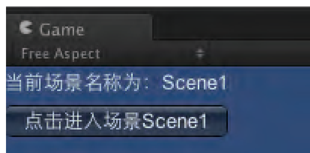


图8-4　切换关卡

代码清单8-5　Main0.cs文件

```
using UnityEngine;
using System.Collections;

public class Main0 : MonoBehaviour {


    void OnGUI()
    {

        GUILayout.Label("当前场景名称为："+Application.loadedLevelName);
        if(GUILayout.Button("点击进入场景Scene1"))
        {
            Application.LoadLevel ("Scene1");
        }
    }
}
```

上述代码使用`Application.LoadLevel()`方法读取新关卡后立即切换，其参数为所读取的新关卡的名称。

8.3.3　截屏

Unity支持截屏操作，用户可以通过它在游戏运行的任意时刻将当前屏幕中的内容以图片的形式保存至本地。目前，很多游戏都有“分享”功能，其作用就是将截图分享到微博或Facebook等社交网站中。截屏操作需调用方法`Application.CaptureScreenshot("name.png")`，该方法的参数为截屏图片的文件名称，成功截屏后该文件将被默认保存在根目录下。本例代码如代码清单8-6所示。

代码清单8-6 Script_08_06.cs文件

```
using UnityEngine;
using System.Collections;
```

```
public class Script_08_06 : MonoBehaviour {

    void OnGUI()
    {

        GUILayout.Label("当前场景名称为: "+Application.loadedLevelName);

        if(GUILayout.Button("点击按钮截图"))
        {
            Application.CaptureScreenshot("name.png");
        }
    }

}
```

如图8-5所示，截屏图片已经顺利保存在工程根目录下，双击即可直接查看效果。

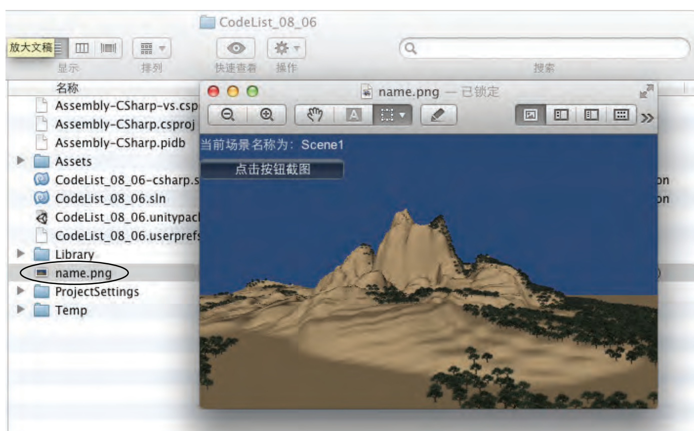


图8-5　截屏

8.3.4　打开网页

此外，Unity还支持在当前设备中调用浏览器打开网页，这里通过`Application.OpenURL("http://blog.csdn.net/xys289187120/")`方法来实现，其参数为待打开网页的完整网址。如图8-6所示，程序使用此方法成功地打开了一个网页，具体代码如代码清单8-7所示。



图8-6　打开网页

代码清单8-7　Script_08_07.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_08_07 : MonoBehaviour {

    void OnGUI()
    {
        GUILayout.Label("当前场景: "+Application.loadedLevelName);
        if(GUILayout.Button("打开网页"))
        {
            Application.OpenURL ("http://blog.csdn.net/xys289187120/");
        }
    }

}
```

8.3.5　退出游戏

在程序中直接调用Application.Quit()方法即可退出当前游戏。需要注意的是，该方法只能在真实设备中执行，而无法在模拟器中执行。

8.4　资源数据库

资源数据库主要用于操作Project视图中的资源文件。它比较灵活，可以使用程序动态增加、删除、修改和查询资源。比如在游戏中联网下载资源包时，它会先将所需的资源下载至本地，然后再使用资源数据将其移动到对应的资源文件夹中，最后使用程序读取对应的游戏资源。

8.4.1　加载资源

用户可以使用资源数据库，在程序中动态加载Project视图中任意类型的资源对象。如图8-7所示，本例使用资源数据库动态加载了一张贴图并将其显示在屏幕中，具体代码如代码清单8-8所示。



图8-7　打开网页

代码清单8-8 Script_08_08.cs文件

```
using UnityEngine;
using System.Collections;
using UnityEditor;


public class Script_08_08 : MonoBehaviour {

    //贴图
    Texture2D texture;

    void Start()
    {
        //从Project视图中加载资源
        texture = (Texture2D)AssetDatabase.LoadAssetAtPath("Assets/Texture/0.png",
            typeof(Texture2D));
    }
    void OnGUI()
    {
        //渲染资源
        GUI.DrawTexture(new Rect(0,0,texture.width,texture.height),texture);

    }

}
```

本例使用资源数据库`LoadAssetAtPath()`方法来读取本地资源，该方法的第一个参数表示欲读取资源的完整路径，第二个参数表示该资源的类型。这里读取了一张PNG图片，并将其赋予`Texture`（纹理）对象，最后使用`OnGUI()`方法将该纹理对象绘制在屏幕中。

8.4.2 创建资源

资源数据库提供了创建资源的接口，可以在程序中动态创建任意资源，并且这些资源一旦创建成功，就不会因为游戏关闭而消失。如图8-8所示，本例动态创建了一个材质，并且将贴图赋予该材质，最后再将该材质附加给立方体游戏对象，具体代码如代码清单8-9所示。

代码清单8-9 Script_08_09.cs文件

```
using UnityEngine;
using System.Collections;
using UnityEditor;

public class Script_08_09 : MonoBehaviour
{

    //贴图对象
    Texture2D texture = null;
    void Start()
    {
        //创建一个材质为默认着色器
        Material mat = new Material (Shader.Find("Transparent/Diffuse"));
```

```
        //在Project视图中加载贴图资源
        texture = (Texture2D)AssetDatabase.LoadAssetAtPath("Assets/Texture/0.png",
            typeof(Texture2D));
        //将贴图资源赋予创建的材质
        mat.mainTexture = texture;
        //将材质添加到Project视图中
        AssetDatabase.CreateAsset(mat, "Assets/mat.mat");
        //创建一个立方体对象
        GameObject objCube = GameObject.CreatePrimitive(PrimitiveType.Cube);
        //将创建的材质赋予此立方体对象
        objCube.renderer.material = mat;
    }
}
```



图8-8　创建资源

以上代码在开始时使用`new Material()`方法创建了游戏材质，该方法的参数为材质的类型，然后使用`CreateAsset()`方法将代码中创建的材质移动到Project视图当中，其中`CreateAsset()`方法的第一个参数表示资源本身，第二个参数表示资源的存放路径，最后再将该材质赋予立方体对象。

8.4.3　创建文件夹

为了方便项目管理，通常会将资源数据进行分类，把相同类型的数据放在同一个文件夹下。资源数据库提供动态创建文件夹的功能。如图8-9所示，在游戏视图中点击“添加一个新文件夹”按钮后，系统会在右侧的Project视图中动态添加文件夹，并且将新创建的材质添加进去，本例

代码如代码清单8-10所示。

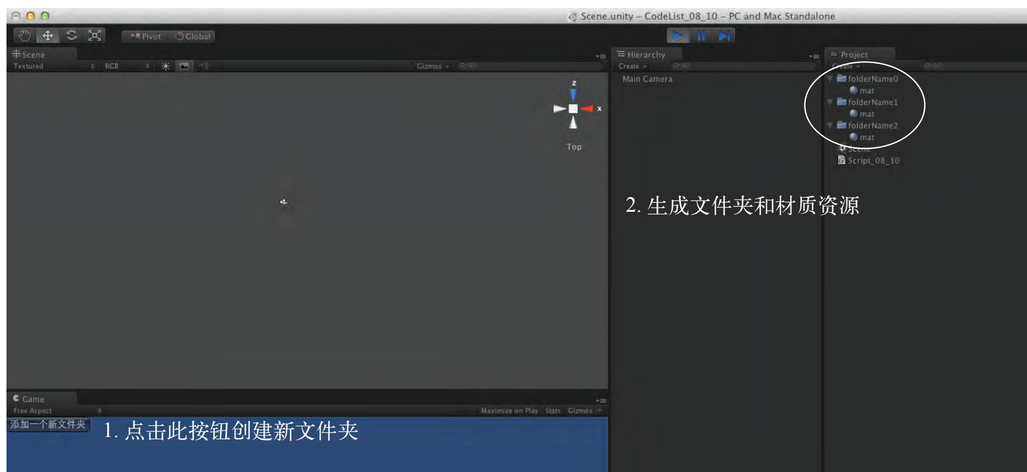


图8-9 创建文件夹

代码清单8-10 Script_08_10.cs文件

```
using UnityEngine;
using System.Collections;
using UnityEditor;

public class Script_08_10 : MonoBehaviour
{
    //记录添加文件夹的次数
    int addId = 0;
    void OnGUI()
    {
        if(GUILayout.Button("添加一个新文件夹"))
        {
            //得到当前文件夹的名称
            string folder  = "folderName"+addId;
            //创建一个新文件夹
            AssetDatabase.CreateFolder("Assets", "folderName"+addId);
            //创建一个材质为默认着色器
            Material  mat = new Material (Shader.Find("Transparent/Diffuse"));
            //将材质添加至新文件夹
            AssetDatabase.CreateAsset(mat, "Assets/"+folder+"/mat.mat");

            addId++;
         }
    }
}
```

本例使用资源数据库的`CreateFolder()`方法来创建文件夹，其中该方法的第一个参数表示文件夹的存放路径，第二个参数表示文件夹的名称，然后使用`CreateAsset()`方法创建一个材

质，并且将该材质放入刚刚创建的文件夹中。

8.4.4　移动与复制

资源数据库提供了移动与复制文件的功能。如图8-10所示，本例将图中材质“folderName0/mat0”与“folderName1/mat1”分别移动并复制到文件夹“folderName2”中，具体代码如代码清单8-11所示。

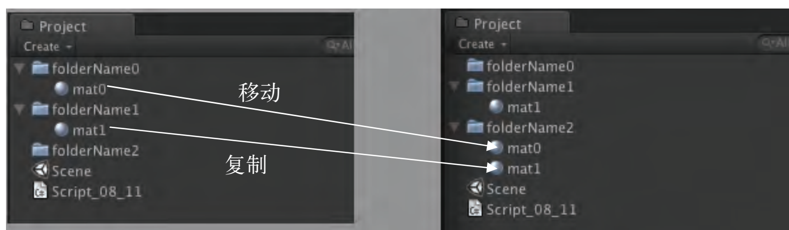


图8-10　创建文件夹

代码清单8-11　Script_08_11.cs文件

```
using UnityEngine;
using System.Collections;
using UnityEditor;

public class Script_08_11 : MonoBehaviour
{
    void Start()
    {
        //复制资源
        AssetDatabase.CopyAsset("Assets/folderName1/mat1.mat",
            "Assets/folderName2/mat1.mat");
        //移动资源
        AssetDatabase.MoveAsset("Assets/folderName0/mat0.mat",
            "Assets/folderName2/mat0.mat");
    }
}
```

以上代码使用资源数据库的`CopyAsset()`方法进行资源的复制，其中该方法的第一个参数表示待复制资源的路径，第二个参数表示复制后的资源路径。另外，使用`MoveAsset()`方法可对资源进行移动，该方法的第一个参数表示待移动资源的路径，第二个参数表示移动后的资源路径。

8.4.5　删除与刷新

资源数据库提供了删除功能，用于删除Project视图中的任意资源文件夹或文件。下列代码使用资源数据库的`DeleteAsset()`方法进行资源的删除，该方法的参数表示待删除资源的完整路径。删除完成后，使用`AssetDatabase.Refresh()`方法刷新Project视图，即可看到删除后的效

果。本例代码如代码清单8-12所示。

代码清单8-12　Script_08_12.cs文件

```
using UnityEngine;
using System.Collections;
using UnityEditor;

public class Script_08_12 : MonoBehaviour
{
    void Start()
    {
        //删除资源
        AssetDatabase.DeleteAsset("Assets/folderName2/mat1.mat");
        //刷新资源视图
        AssetDatabase.Refresh();
    }
}
```

8.4.6　实例——鼠标拖动模型

在学习本例之前，我们先来了解一下处理鼠标事件的6大方法，这些方法都是由脚本继承的父类MonoBehaviour实现的，只需要在脚本中监听其方法即可，具体代码如代码清单8-13所示。

代码清单8-13　Script_08_13.cs文件

```
using UnityEngine;
using System.Collections;
using UnityEditor;

public class Script_08_13 : MonoBehaviour
{

    void OnMouseDrag()
    {
        Debug.Log("鼠标拖动该模型区域时");
    }

    void OnMouseDown()
    {
        Debug.Log("鼠标按下时");
    }
    void OnMouseUp()
    {
        Debug.Log("鼠标抬起时");
    }

    void OnMouseEnter()
    {
        Debug.Log("鼠标进入该对象区域时");
    }
    void OnMouseExit()
```

```
        {
            renderer.material.color = Color.white;
            Debug.Log("鼠标离开该模型区域时");
        }
        void OnMouseOver()
        {
            renderer.material.color = Color.red;
            Debug.Log("鼠标停留在该对象区域时");
        }
    }
```

和往常一样，我们将脚本绑定在摄像机中，在运行游戏后，问题就出现了——上述代码中所有处理鼠标事件的方法都未能执行，这是为什么呢？因为这些方法只有在鼠标接触到绑定该脚本的对象时才会触发，而摄像机又是一个特殊的对象，鼠标永远不可能接触到摄像机对象。因此，我们可以在游戏世界中创建一个任意模型，并将这段代码绑定在该模型中，然后运行游戏，操作鼠标，即可在脚本中监听到鼠标的所有事件。

下面我们使用`OnMouseDrag()`方法来拖曳屏幕中的模型。如图8-11所示，将鼠标放置在该立方体对象上，当在“上”、“下”、“左”、“右”4个方向上拖动鼠标时，立方体对象会随之移动。



图8-11　拖动鼠标

本例将使用资源数据库来创建一个立方体对象，并且将材质与贴图赋予该对象，具体代码如代码清单8-14所示。

代码清单8-14　Script_08_14.cs文件

```
using UnityEngine;
```

```
using System.Collections;
using UnityEditor;

public class Script_08_14 : MonoBehaviour
{

    //贴图对象
    Texture2D texture = null;
    void Start()
    {
        //创建一个材质为默认着色器
        Material mat = new Material (Shader.Find("Transparent/Diffuse"));
        //在Project视图中加载贴图资源
        texture = (Texture2D)AssetDatabase.LoadAssetAtPath("Assets/Texture/0.png",
            typeof(Texture2D));
        //将贴图资源赋予所创建的材质
        mat.mainTexture = texture;
        //将材质添加至Project视图
        AssetDatabase.CreateAsset(mat, "Assets/mat.mat");
        //创建一个立方体对象
        GameObject objCube = GameObject.CreatePrimitive(PrimitiveType.Cube);
        //将所创建的材质赋予这个立方体对象
        objCube.renderer.material = mat;
        //最后将处理鼠标事件的脚本添加至该立方体汇总
        objCube.AddComponent("Move");
    }

}
```

本例使用资源数据库创建了一个普通的立方体对象。为了在脚本中监听鼠标的事件，需要将脚本绑定在该对象中，并使用该脚本的对象调用`AddComponent()`方法，该方法的参数为脚本的名称。至此，脚本组件绑定完成。

处理鼠标事件的代码如代码清单8-15所示。

代码清单8-15 Move.cs文件

```
using UnityEngine;
using System.Collections;

public class Move : MonoBehaviour {

    void OnMouseDrag()
    {
        Debug.Log("鼠标拖动该模型区域时");
        transform.position += Vector3.right * Time.deltaTime*Input.GetAxis ("Mouse
            X");
        transform.position += Vector3.up * Time.deltaTime*Input.GetAxis ("Mouse Y");
    }

    void OnMouseDown()
    {
        Debug.Log("鼠标按下时");
    }
```

8

```
    void OnMouseUp()
    {
        Debug.Log("鼠标抬起时");
    }

    void OnMouseEnter()
    {
        Debug.Log("鼠标进入该对象区域时");
    }
    void OnMouseExit()
    {
        renderer.material.color = Color.white;
        Debug.Log("鼠标离开该模型区域时");
    }
    void OnMouseOver()
    {
        renderer.material.color = Color.red;
        Debug.Log("鼠标停留在该对象区域时");
    }

}
```

本例主要实现了鼠标在屏幕中对模型的拖曳，所以需要监听鼠标点中该模型时的事件，这里OnMouseDrag()方法即可监听到鼠标点中立方体对象的事件，然后获取鼠标当前的x轴与y轴坐标并动态地修改立方体的位置，实现鼠标拖曳模型的效果。

8.4.7 实例——鼠标拣选

不论是2D游戏还是3D游戏，其映射给玩家的永远是一个2D平面，此时鼠标与游戏之间的交互就存在一个问题，即鼠标在2D屏幕中选择的点如何映射到3D游戏中？可以使用上一章所介绍的射线原理来解决此问题，以摄像机的位置为原点，以鼠标在屏幕中选择的当前点为目标点，发射一条射线，获取这条射线终点的三维坐标系即可。

鼠标拣选原理的应用非常广泛，比如当玩家在地图中选择一个点时，人物会朝该点移动。本例将模拟完成鼠标拣选的过程。如图8-12所示，当鼠标在游戏地形中选择一个点后，系统就立即在该点创建一个立方体对象。此时需要用程序来检测该点是否处于游戏地形中，若否，则不创建立方体对象。处理鼠标事件的代码如代码清单8-16所示。

代码清单8-16 Script_08_15.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_08_15 : MonoBehaviour {

    void Update()
    {
        //点击鼠标左键后
        if(Input.GetMouseButtonDown(0))
        {
```

```
                //创建一条从摄像机到鼠标选择的当前点的射线
                Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
                RaycastHit hit;
                //计算这条射线是否发射到地形中
                if (Physics.Raycast(ray, out hit))
                {
                    //确定选择的点为地形后，创建立方体对象
                    GameObject objCube = GameObject.CreatePrimitive(PrimitiveType.Cube);
                    objCube.transform.position = hit.point;

                }
            }
        }
    }
```

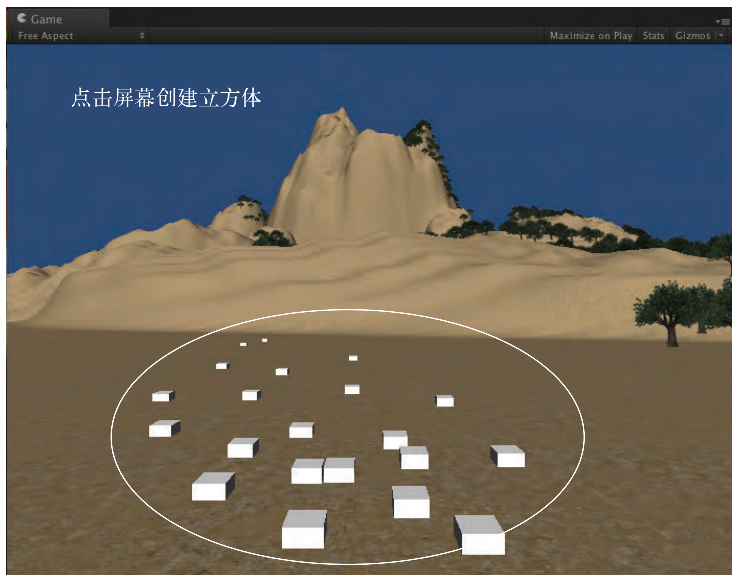


图8-12　鼠标拣选

当鼠标在3D游戏世界中选中一点时，系统会使用`Camera.main.ScreenPointToRay()`方法由摄像机的位置向鼠标选中时3D世界中的当前位置发射一条射线，然后使用`Physics.Raycast()`方法计算出该射线与地形相交点的位置，最后在该点创建一个立方体对象。

8.5　游戏实例——接受任务

本例将使用游戏持久化数据以及鼠标拣选实现一个简单的游戏任务系统。如图8-13所示，玩家可先点击NPC接受任务，然后使用鼠标点击寻找“红宝石”，在收集到20块红宝石后向NPC提交任务，即可完成游戏。



图8-13　接受任务

如果游戏中途退出，系统会自动保存玩家当前获得的红宝石数量，并在下次进入游戏时读取上一次保存的游戏数据，继续游戏。如图8-14所示，目前任务已经完成，点击NPC结束该任务。



图8-14　完成任务

选择NPC接受任务的代码为Npc.cs脚本，具体如代码清单8-17所示。

代码清单8-17 Npc.cs文件

```
using UnityEngine;
using System.Collections;

public class Npc : MonoBehaviour {

    //游戏状态机

    //默认状态
    public const int TASK_STATE_DEFAULT = 0;
    //信息提示状态
    public const int TASK_STATE_OVER = 1;
    //打开任务窗口状态
    public const int TASK_STATE_OPEN = 2;
    //游戏状态
    public const int TASK_STATE_GAME = 3;
    //游戏完成状态
    public const int TASK_STATE_COMPLETE = 4;
    //游戏结束状态
    public const int TASK_STATE_FINSH = 5;

    //当前游戏状态
    public int gameState = 0;

    //脚本对象
    Script_08_16 script = null;

    //任务窗口显示区域
    Rect windowRect  = new Rect(100,0,200,100);

    void Start()
    {
        //得到脚本对象
        script = GameObject.Find("/Main Camera").GetComponent<Script_08_16>();
        //在游戏存档中读取游戏状态数据
        gameState = PlayerPrefs.GetInt("gameState",TASK_STATE_DEFAULT);
    }

    void OnMouseDown()
    {

        if(gameState == TASK_STATE_OVER)
        {
            //进入弹出任务窗口状态
            gameState = TASK_STATE_OPEN;
        }else
        if(gameState ==TASK_STATE_COMPLETE)
        {
            //进入任务完成状态
            gameState = TASK_STATE_FINSH;
```

```
        }
    }

    void OnMouseOver()
    {
        if(gameState == TASK_STATE_DEFAULT)
        {
            //进入提示信息状态
            gameState = TASK_STATE_OVER;
        }
    }



    void OnGUI()
    {
        //状态机的绘制
        switch(gameState)
        {
            case TASK_STATE_OVER:
                GUILayout.Box("点击NPC接受一个任务");
            break;

            case TASK_STATE_OPEN:
                windowRect = GUILayout.Window (0, windowRect, AddWindow, "接受任务");
            break;
            case TASK_STATE_GAME:
                GUILayout.Box("目前收集到的宝石数量为："+script.items);
            break;
            case TASK_STATE_COMPLETE:
                GUILayout.Box("已经收集到10颗宝石，请到NPC处交任务，宝石数量不再累加！");
            break;
            case TASK_STATE_FINSH:
                GUILayout.Box("任务完成");
                if(GUILayout.Button("重新开始"))
                {
                    //重置游戏
                    gameState = TASK_STATE_DEFAULT;
                    script.items = 0;
                    PlayerPrefs.SetInt("itemCount", script.items);
                    PlayerPrefs.SetInt("gameState", gameState);
                }
            break;
        }


    }

    void AddWindow(int windowID)
    {

        GUILayout.Label("请在游戏中寻找20块红宝石");
        //开始一个水平布局
        GUILayout.BeginHorizontal();
```

```
            if (GUILayout.Button ("接受任务"))
            {
                gameState = TASK_STATE_GAME;
            }

            if (GUILayout.Button ("拒绝任务"))
            {
                gameState = TASK_STATE_DEFAULT;
            }
            //关闭水平布局
            GUILayout.EndHorizontal();

        }
    }
```

本段脚本运用游戏状态机的原理，使用整型变量gameState记录当前任务的状态并通过OnGUI()方法更新状态。

使用鼠标拾取红宝石的代码如代码清单8-18所示。

代码清单8-18 Script_08_16.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_08_16 : MonoBehaviour {

    //红宝石数量
    public int items;
    //脚本对象
    private Npc npc = null;

    void Start()
    {
        //在游戏存档中读取上次保存游戏时红宝石的数量
        items =  PlayerPrefs.GetInt("itemCount",0);
        //获取脚本对象
        npc = GameObject.Find("/NPC").GetComponent<Npc>();

    }


    void Update()
    {
        //状态是否为游戏中
        if(npc.gameState == Npc.TASK_STATE_GAME)
        {
            //是否点击鼠标左键
            if(Input.GetMouseButtonDown(0))
            {
                //创建鼠标点击的射线
                Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
                RaycastHit hit;
```

```
                    //是否选中游戏对象
                    if (Physics.Raycast(ray, out hit))
                    {
                        //得到选中的游戏对象
                        GameObject obj = hit.collider.gameObject;
                        //是否为红宝石对象
                        if(obj.name == "cube")
                        {
                            //红宝石数量加1
                            items++;
                            if(items>=10)
                            {
                              //当红宝石数量达到10时，表示游戏完成
                              npc.gameState = Npc.TASK_STATE_COMPLETE;
                            }
                            //销毁红宝石对象
                            Destroy(obj);
                            //保存红宝石数量以及游戏状态
                            PlayerPrefs.SetInt("itemCount", items);
                            PlayerPrefs.SetInt("gameState", npc.gameState);
                        }
                    }
                }
            }
        }
    }
```

拾取红宝石同样运用了射线原理，当射线与游戏对象相交时，首先判断该游戏对象是否为红宝石对象，使用hit.collider.gameObject引用即可得到相交的游戏对象。如果该对象是“红宝石”则使用Destroy()方法销毁它，并递增玩家背包中的红宝石数量。

8.6　本章小结

本章在开始部分介绍了如何使用PlayerPrefs类持久化存储数据，接着讲述了自定义文件的创建过程，以及使用流对自定义文件进行读取与写入的方法，通过实例向读者展示了在本地持久化存储数据的过程；然后介绍了应用程序、游戏场景与关卡的相关知识，以及如何使用代码来切换当前游戏关卡，另外还介绍了如何在程序中截屏与调用平台浏览器访问互联网等；接着介绍了资源数据库中增加、删除、修改、查询资源的过程，以及鼠标拣选与鼠标事件，最后以“接受任务”实例结束本章。

第9章

多媒体与网络

多媒体是游戏中必不可少的元素之一，而游戏中的多媒体主要包括音频和视频。相对而言，网络则是一个比较复杂的元素，涵盖的知识点较为广泛。一般来说工程的相关资源文件都可以通过下载最终展示在游戏当中。例如，我们可以将视频或音频文件上传至服务器端，然后Unity通过访问服务器端即可下载并播放这些文件。本章将重点介绍客户端与服务器端之间的交互。

9.1 游戏音频

游戏音频在游戏开发中占据着重要的地位，优秀的音乐与音效可以提升游戏的整体效果。总的来说，音频可以分为两种，一种为游戏音乐，另一种为游戏音效，前者多为较长的音乐，如游戏背景音乐；而后者则多是较短的音效，如开枪或打怪物时“砰砰”的游戏音效。本节将重点剖析Unity 3D游戏音乐与音效的播放。

9.1.1 音频介绍

Unity 3D游戏引擎共支持4种音乐格式的文件，具体如下。

- aiff：适用于较短的音乐文件，可用作游戏音效。
- wav：适用于较短的音乐文件，可用作游戏音效。
- mp3：适用于较长的音乐文件，可用作游戏音乐。
- ogg：适用于较长的音乐文件，可用作游戏音乐。

9.1.2 添加音频

音频是一个游戏组件，因此需要将其绑定在游戏中的某个对象上才能发挥作用。添加音频组件的方法如下，首先在Unity菜单栏中选择“GameObject”→“CreateEmpty”菜单项，创建一个空游戏对象，然后在Hierarchy视图中选择该对象，接着在导航菜单栏中选择“Component”→“Audio”→“Audio Source”菜单项，如图9-1所示，此时音频组件将被绑定在该对象上。

本例中，我们首先将“0.mp3”文件拖曳到Project资源视图中，并将音乐文件放置在audio文件夹内，如图9-2所示。然后选择已绑定音频组件的游戏对象，此时右侧的Inspector视图中将出现该对象的音频组件信息。由于目前音频组件中尚没有可以播放的音频文件，因此需要对其赋值。

按照图9-2中箭头的指示方向将声音文件拖曳至右侧的Audio Clip（音频剪辑），这样就完成了音频组件的赋值。

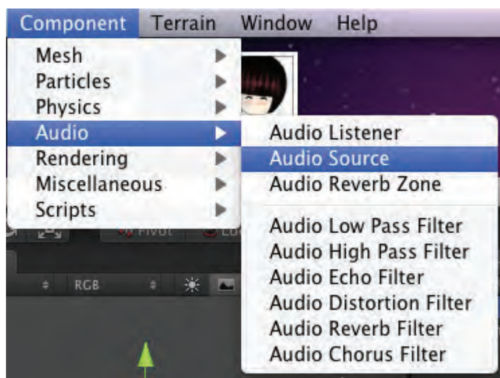


图9-1　音频组件

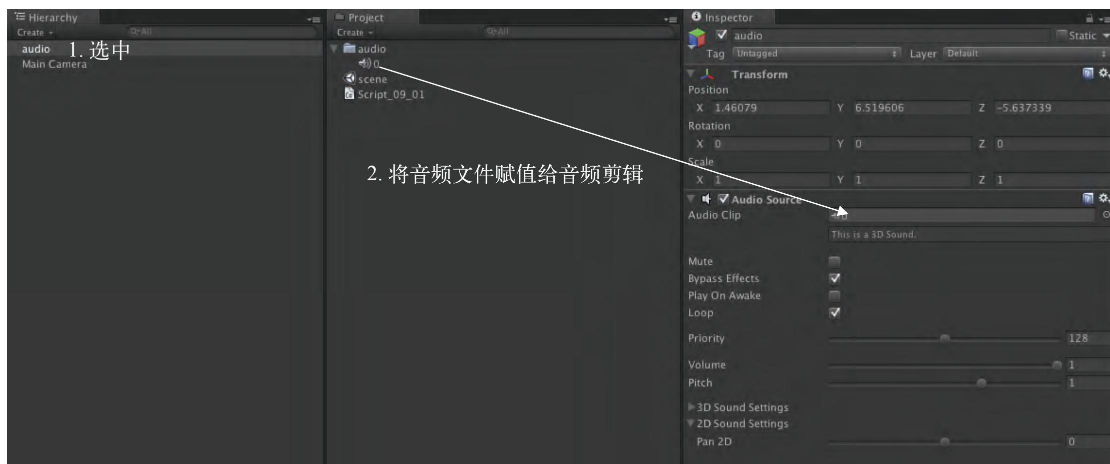


图9-2　绑定组件

此时，可以在右侧的Inspector视图中看到音频组件所包含的属性，通过调节这些属性可以使音乐文件更加完善。下面简要列举其中各个属性的含义。

- Audio Clip：声音片段，可对其直接赋值或在代码中动态地截取音乐文件。
- Mute：是否静音。
- Bypass Effects：是否打开音频特效。
- Play On Awake：开机自动播放。
- Loop：循环播放。
- Priority：音频播放的优先级，0表示优先级最高，256表示优先级最低，默认数值为128。

- Volume：音量大小，取值范围为0.0 ~ 1.0。
- Pitch：播放速度，取值范围在–3 ~ 3之间，设置为1表示正常播放，小于1表示慢速播放，大于1表示加速播放。

需要注意的是，必须在Main Camera中勾选“Audio Listener”组件。默认情况下，该组件被绑定在主摄像机中，如图9-3所示。此外，用户也可以创建一个游戏对象来绑定该组件，否则Unity将无法播放音频文件。

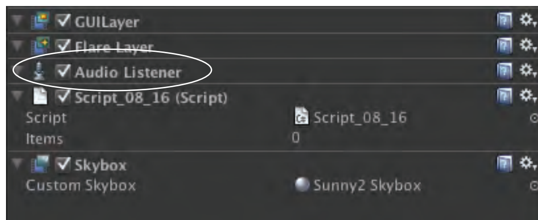


图9-3 音频监听

9.1.3 播放音频

系统在播放音频时，需要使用AudioSource组件对象，具体的操作方法是在Hierarchy视图的编辑器中将已绑定音频组件的游戏对象拖曳赋值给脚本中的“AudioSource”对象。然后，点击“播放音乐”、“停止音乐”或“暂停音乐”按钮，即可实现相应的效果，还可以使用滑动条来控制音乐的音量，如图9-4所示，具体代码如代码清单9-1所示。



图9-4 播放音频

9

代码清单9-1 Script_09_01.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_09_01 : MonoBehaviour
{

    //音乐文件
    public AudioSource music;
    //音量
    public float musicVolume;

    void Start()
    {
         //设置默认音量
         musicVolume = 0.5F;
    }
    void OnGUI()
    {

        //播放音乐按钮
        if (GUI.Button(new Rect(10, 10, 100, 50), "播放音乐"))
        {

            //没有播放中
            if (!music.isPlaying)
            {
                //播放音乐
                music.Play();
            }

        }

        //关闭音乐按钮
        if (GUI.Button(new Rect(10, 60, 100, 50), "停止音乐"))
        {

            if (music.isPlaying)
            {
                //关闭音乐
                music.Stop();
            }
        }
        //暂停音乐
        if (GUI.Button(new Rect(10, 110, 100, 50), "暂停音乐"))
        {
            if (music.isPlaying)
            {
                //暂停音乐
                music.Pause();
            }
        }
```

```
            //创建一个用于动态修改音乐音量的横向滑动条
            musicVolume = GUI.HorizontalSlider (new Rect(160, 10, 100, 50), musicVolume,
                0.0F, 1.0F);

            //将音量的百分比显示出来
            GUI.Label(new Rect(160, 50, 300, 20), "游戏音量" + (int)(musicVolume * 100) +
                "%");

            if (music.isPlaying)
            {
                //在播放音乐中设置音乐音量，其取值范围为0.0F～1.0F
                music.volume = musicVolume;
            }
        }
    }
```

在上述代码中，AudioSource对象可使用Play()、Stop()和Pause()方法直接控制音乐的播放、停止或暂停。此外，我们还创建了一个水平滑动条，拖动滑块即可改变音量的大小。

9.2 游戏视频

视频元素能够丰富游戏的效果，一般情况下，大型3D游戏都会选择游戏中的精彩视频作为开场动画，这会给首次进入游戏的玩家带来眼前一亮的感觉。在Unity中，我们需要使用MovieTexture（电影纹理）来添加游戏视频。MovieTexture对象继承自纹理对象，所以其用法与纹理基本一样。Unity支持的视频格式包括.mov、.mpg、.mpeg、.mp4、.avi和.asf。本节将详细讨论游戏视频的制作与播放。

9.2.1 创建视频

在播放视频之前，需要先创建电影纹理。直接将视频文件拖入Project视图中，系统会自动生成对应的电影纹理。本例中，我们只需将“0.mp4”文件拖曳到Project视图中，系统就会自动生成电影纹理资源，如图9-5所示。可以在右侧的Inspector视图设置视频的相关属性，其中各个属性的含义如下。

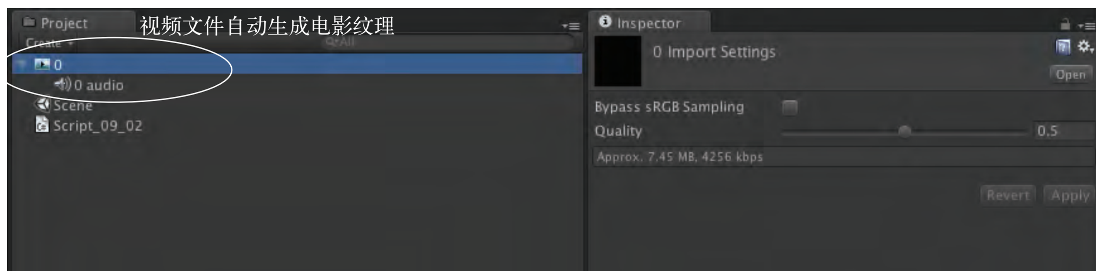


图9-5 电影纹理

9

- ❑ Aniso Level：播放等级，直接影响视频播放的效果。
- ❑ Filter Mode：视频播放模式。
- ❑ Loop：是否循环播放。
- ❑ Quality：视频原始资源压缩比例。

9.2.2　播放视频

和音频组件一样，电影纹理也需要绑定在某个游戏对象上才能发挥其效果。一般情况下，游戏视频都是以平面形式展现的，所以可在游戏场景中创建一个“Plane”（面）对象。接下来，将 Script_09_02脚本绑定在Plane对象上，再将视频文件赋值给Script_09_02脚本中的电影纹理对象，如图9-6所示。

播放视频时，应使主摄像机照射在面的中央。由于默认情况下3D世界比较黑暗，所以可以在其中添加一些灯光。本例使用了平行光，它和主摄像机同时照射在这个面的中央。

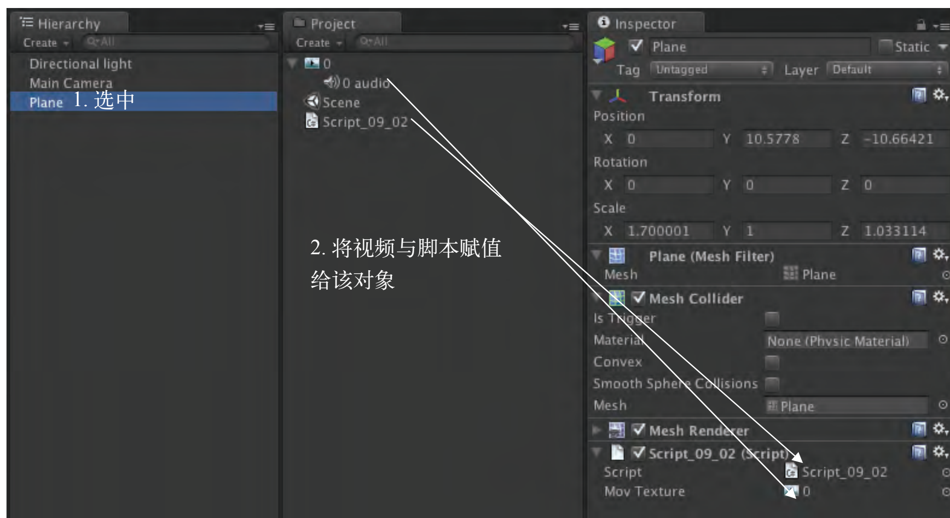


图9-6　绑定对象

点击“播放/继续”、“暂停播放”、“停止播放”按钮，即可完成视频的一些基本操作，如图9-7所示，具体代码如代码清单9-2所示。

代码清单9-2　Script_09_02.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_09_02: MonoBehaviour {

    //电影纹理
    public MovieTexture movTexture;
```

```
    void Start()
    {
        //设置当前对象的主纹理为电影纹理
        renderer.material.mainTexture = movTexture;
        //设置电影纹理播放模式为循环
        movTexture.loop = true;
    }

    void OnGUI()
    {
        if(GUILayout.Button("播放/继续"))
        {
            //播放/继续播放视频
            if(!movTexture.isPlaying)
            {
                movTexture.Play();
            }

        }

        if(GUILayout.Button("暂停播放"))
        {
            //暂停播放
            movTexture.Pause();
        }

        if(GUILayout.Button("停止播放"))
        {
            //停止播放
            movTexture.Stop();
        }
    }
}
```

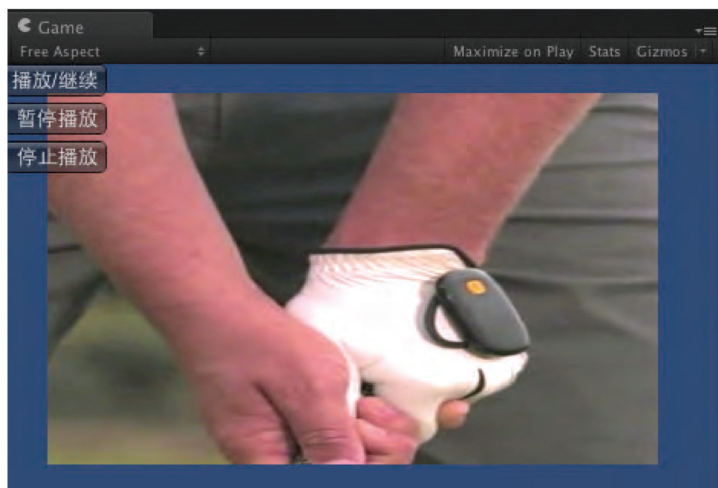


图9-7　播放视频

由于Script_09_02脚本绑定在“Plane”对象上，为了在对象中显示视频，需要将Project视图中的电影纹理资源对象赋值给脚本，然后电影纹理对象使用Play()、Pause()或Stop()方法即可播放、暂停或结束视频。

9.2.3　GUI播放视频

电影纹理不仅可以在对象中播放，也可以在GUI中播放，不过在GUI中播放的效率要比在游戏对象中低一些，但是在GUI中可随意修改视频的尺寸。本例使用GUI绘制了一个全屏的游戏视频，将代码清单9-3所示的脚本绑定在摄像机中即可看到效果，如图9-8所示。

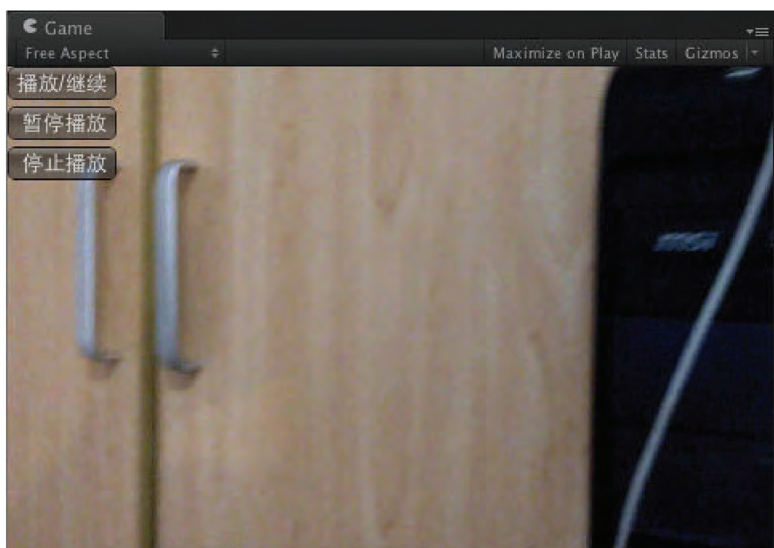


图9-8　播放视频

代码清单9-3　Script_09_03.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_09_03 : MonoBehaviour {

    //电影纹理
    public MovieTexture movTexture;

    void Start()
    {
        //设置电影纹理的播放模式为循环
        movTexture.loop = true;
    }
```

```
    void OnGUI()
    {
        //绘制电影纹理
        GUI.DrawTexture (new Rect (0,0, Screen.width, Screen.height),movTexture,
            ScaleMode.StretchToFill);

        if(GUILayout.Button("播放/继续"))
        {
            //播放/继续播放视频
            if(!movTexture.isPlaying)
            {
                movTexture.Play();
            }

        }

        if(GUILayout.Button("暂停播放"))
        {
            //暂停播放
            movTexture.Pause();
        }

        if(GUILayout.Button("停止播放"))
        {
            //停止播放
            movTexture.Stop();
        }
    }
}
```

在上述代码中，我们使用GUI.DrawTexture()方法来绘制电影纹理，该方法的第一个参数用于设置电影纹理的绘制区域，可比较灵活地控制视频的尺寸。Plane对象属于网格模型，而网格模型的尺寸是无法轻易修改的，所以在播放视频时，它不如GUI灵活。

9.3 网络

网络下载功能在游戏中已经非常普遍，它就好比游戏对外的接口，可动态拓展游戏的内容，比如下载游戏场景、人物模型、游戏音乐和游戏视频等。资源下载完毕后，可被程序读取并合并到游戏中。

Unity为开发者提供了WWW下载类，它的原理是以GET请求的形式向服务器请求数据，然后等待服务器返回，在向服务器请求数据时，将请求的地址传入其构造函数即可开始下载。在下载过程中，可以使用Yield()方法或者isDone()方法来判断下载是否完成。

9.3.1 下载文件

下载文件时，首先需要知道其下载地址（可以指向本机或网络），然后通过该地址来调用Unity中的下载方法。文件下载成功后，会保存在WWW对象中。使用Unity提供的下载方法，开发者可

以任意下载文件，但需要注意的是，目前系统只支持PNG和JPG类型贴图文件的下载。如果下载失败，系统会抛出异常，并使用`Debug.Log(www.error)`方法打印错误信息。如图9-9所示，本例将使用下载的贴图文件来更换`Plane`对象自身的贴图，具体代码如代码清单9-4所示。

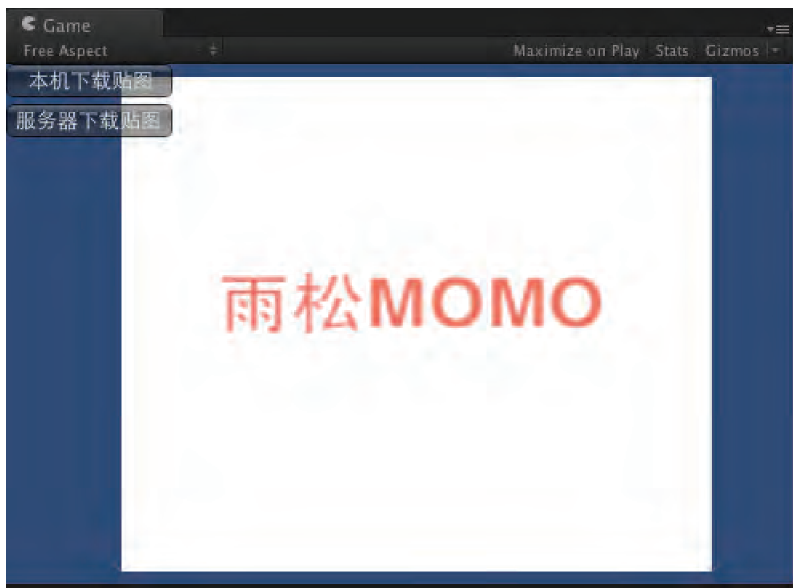


图9-9　下载贴图

代码清单9-4　Script_09_04.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_09_04 : MonoBehaviour
{

    //本机下载的贴图
    private Texture tex0 ;

    //服务器下载贴图
    private Texture tex1;

    IEnumerator loadLocal()
    {
        //本机下载
        if(tex0 == null)
        {
            //资源在本机的路径
            WWW date = new WWW("file://" + Application.dataPath + "/0.png");
            //等待下载完成
            yield return date;
```

```
            //得到下载的贴图
            tex0 = date.texture;
        }
        //更换为下载的贴图
        renderer.material.mainTexture = tex0;

    }

    IEnumerator loadNetWork()
    {
        //服务器网页下载
        if(tex1 == null)
        {
            //资源的url服务器路径
            WWW date = new WWW("http://www.google.com.hk/intl/zh-CN/images/
                logo_cn.png");
            //等待下载完成
            yield return date;
            //得到下载的贴图
            tex1 = date.texture;
        }
        //更换为下载的贴图
        renderer.material.mainTexture = tex1;
    }

     void OnGUI()
    {
         if(GUILayout.Button("本机下载贴图"))
         {
            StartCoroutine(loadLocal());
         }


         if(GUILayout.Button("服务器下载贴图"))
         {
             StartCoroutine(loadNetWork());
         }
    }


}
```

小提示 开发者可以使用WWW类下载资源包、图片、声音和视频等文件，但需要注意的是，所下载的视频与音频文件必须为OGG类型，否则无法播放。

9

9.3.2 自定义资源包

Unity可将游戏对象、材质和贴图等资源文件封装在自定义资源包中。通过数据流就可以取得并且解析该资源包。资源包可以是任意类型的资源，下面我们将学习如何使用Unity编辑器生

成自定义资源包。如图9-10所示，首先选择需要生成的资源文件，这里我们选择了3个材质资源，然后在Unity导航菜单栏中选择“创建资源包”选项生成资源包，最后将生成的资源包放置在“AssetBundles”文件夹中，具体代码如代码清单9-5所示。

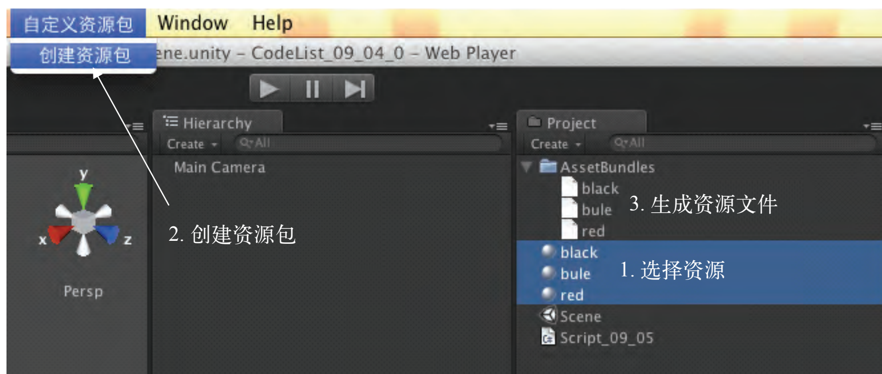


图9-10　创建资源包

代码清单9-5　Script_09_05.cs文件

```
using UnityEngine;
using System.Collections;
using UnityEditor;
using System.IO;

public class Script_09_05 : MonoBehaviour
{

    [MenuItem("自定义资源包/创建资源包")]
    static void ExecCreateAssetBunldes()
    {

        //设置保存资源包的根路径
        string targetDir =  Application.dataPath + "/AssetBundles";

        Object[] SelectedAsset = Selection.GetFiltered(typeof (Object),
            SelectionMode.DeepAssets);

        //创建一个目录，用于保存资源包
        if(!Directory.Exists(targetDir))
        {
            Directory.CreateDirectory(targetDir);
        }

        for(int i = 0; i < SelectedAsset.Length; i++)
        {
```

```
                //生成资源包的完整路径
                string filePath = targetDir + "/"  + SelectedAsset[i].name + ".unity3d";

                //根据路径判断资源包文件是否存在
                if(File.Exists(filePath))
                {
                    //如果存在，直接删除它
                    File.Delete(filePath);
                }
                //生成新的资源包文件
                if(BuildPipeline.BuildAssetBundle(SelectedAsset[i], null, filePath,
                    BuildAssetBundleOptions.CollectDependencies))
                {
                    //表示资源包文件生成成功
                    Debug.Log("资源包文件生成成功");
                    //直接刷新Project视图，查看资源
                    AssetDatabase.Refresh();
                }else
                {
                    //表示资源包文件生成失败
                    Debug.Log("资源包文件生成失败");
                }
            }
        }
    }
```

在上述代码的最后，我们使用方法`BuildPipeline.BuildAssetBundle()`进行生成资源包，该方法的返回值表示资源包是否成功生成。为了避免资源包重复生成，使用`File.Exists()`方法判断该资源包是否存在，如果已存在，则使用方法`File.Delete()`先删除它，然后再生成。

9.3.3 下载资源包

在开发中，我们可将生成的资源包保存在服务器端，然后通过客户端去下载。资源包中保存的资源文件类型多样，使用时需要将资源文件转换成正确的文件类型。下面通过一个示例的形式学习如何下载与使用资源包中的资源文件，如图9-11所示，具体代码如代码清单9-6所示。

代码清单9-6 Script_09_06.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_09_06 : MonoBehaviour
{

    public void OnGUI()
    {
        if(GUILayout.Button("下载红色材质"))
        {
```

```
            StartCoroutine(loadBundleMat("red"));
        }
        if(GUILayout.Button("下载黑色材质"))
        {
            StartCoroutine(loadBundleMat("black"));
        }
        if(GUILayout.Button("下载蓝色材质"))
        {
            StartCoroutine(loadBundleMat("blue"));
        }
        if(GUILayout.Button("下载并创建对象"))
        {
            StartCoroutine(loadBundleObject("Cube"));
        }
    }


    IEnumerator loadBundleMat (string name)
    {
        Material mat;
        //资源在本机的路径
        WWW date = new WWW("file://" + Application.dataPath +
            "/AssetBundles/"+name+".unity3d");
        //等待下载完成
        yield return date;
        //将下载得到的资源文件前置转换成材质资源
        mat = (Material)date.assetBundle.mainAsset;

        //更换为下载的资源
        renderer.material= mat;
        //释放资源对象
        date.assetBundle.Unload(false);
    }

    IEnumerator loadBundleObject (string name)
    {

        //资源在本机的路径
        WWW date = new WWW("file://" + Application.dataPath + "/AssetBundles/"+name+".
            unity3d");
        //等待下载完成，并且克隆游戏对象
        yield return Instantiate(date.assetBundle.mainAsset);
        //释放资源对象
        date.assetBundle.Unload(false);
    }

}
```

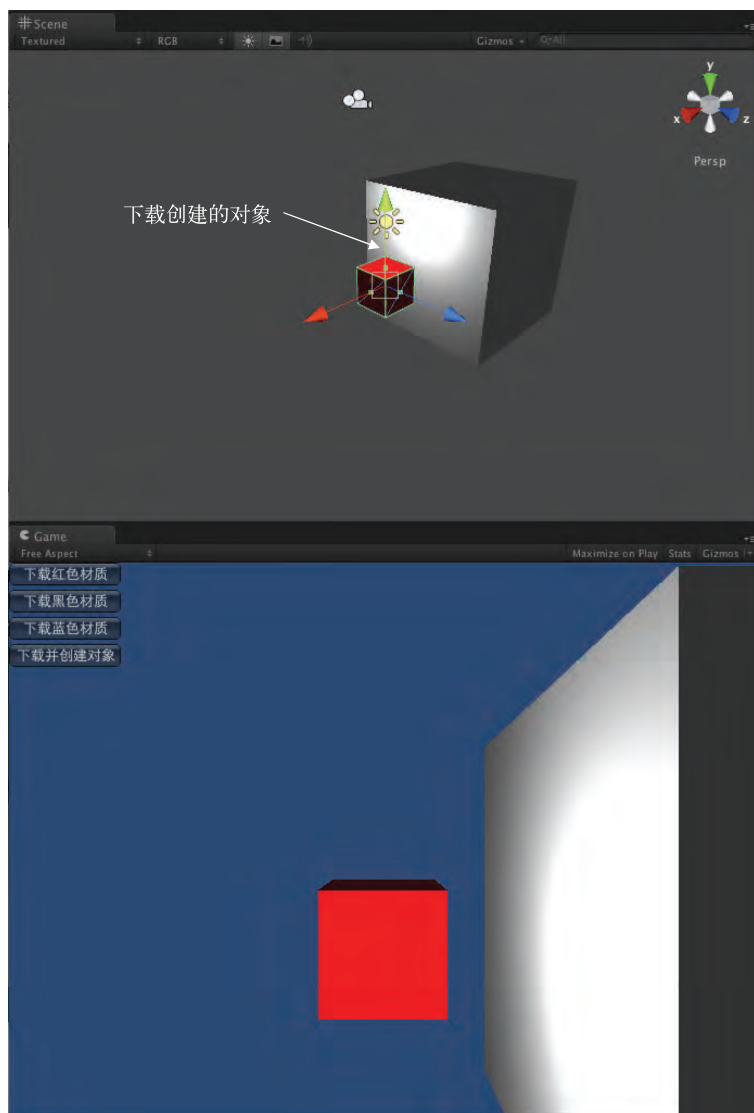


图9-11　下载资源包

资源包下载完毕后，资源文件将保存在`assetBundle.mainAsset`对象中，接着使用该对象即可。由于连续下载资源包对象会出错，所以下载完资源后需要手动释放对象，这可以使用`assetBundle.Unload()`方法完成。

9.3.4　创建本地服务器

客户端与服务器相互连接组成C/S结构。服务器相对独立，可支持多个客户端的访问。两者

之间交互的原理为：客户端向服务器发送请求，服务器给予反馈。这就好比一个班级，学生是客户端，向老师提出问题，而老师则是服务器，可用来回答学生们不同的问题。

在本地创建服务器时，应先判断联网状态，然后再创建服务器连接。开发者可以通过Network.peerType引用获取当前网络的连接状态。网络连接状态分为4种，具体如下。

- NetworkPeerType.Disconnected：未开启状态，在此状态下开始进行网络连接。
- NetworkPeerType.Server：表示成功连接服务器。
- NetworkPeerType.Client：表示成功连接客户端。
- NetworkPeerType.Connecting：表示正在尝试连接。

本例将使用Unity在本地创建一个服务器，具体代码如代码清单9-7所示。

代码清单9-7　Script_09_07.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_09_07 : MonoBehaviour
{

    //端口号
    int Port  = 10000;

    void OnGUI()
    {
        //网络连接状态
        switch(Network.peerType)
        {
            //服务器未开启状态
            case NetworkPeerType.Disconnected:
                StartServer();
            break;
            //成功连接至服务器端
            case NetworkPeerType.Server:
                OnServer();
            break;
            //成功连接至客户端
            case NetworkPeerType.Client:

            break;
            //正在尝试连接
            case NetworkPeerType.Connecting:

            break;
        }
    }

    //创建本机服务器
```

```
    void StartServer()
    {
        if(GUILayout.Button("创建本机服务器"))
        {
            //创建服务器，允许10台主机连接
            NetworkConnectionError error = Network.InitializeServer(10, Port, false);
            //如果连接失败，将错误信息打印出来
            Debug.Log("连接状态："+error);
        }
    }

    void OnServer()
    {
         GUILayout.Label("服务器创建完毕，等待客户端连接");
         //得到客户端连接的数量
         int length = Network.connections.Length;
         for(int i =0; i<length; i++)
         {
             GUILayout.Label("连接服务器客户端ID" + i);
             GUILayout.Label("连接服务器客户端IP" + Network.connections[i].ipAddress);
             GUILayout.Label("连接服务器客户端端口号" + Network.connections[i].port);
         }
         //断开服务器
         if (GUILayout.Button("断开服务器"))
         {
             Network.Disconnect();
         }


    }

}
```

本例在NetworkPeerType.Disconnected状态下使用Network.InitializeServer()方法创建了服务器端。Network.InitializeServer()方法共有3个参数，其中第一个参数表示可连接到该服务器端的客户端数量，第二个参数表示服务器的端口号，第三个参数表示是否支持Nat方式的连接。断开连接时，只需调用Network.Disconnect()方法即可。

由于在Unity中不能同时打开两个项目，所以无法在编辑阶段进行测试。为了测试这段程序，我们先将工程打包。在Unity导航菜单栏中选择“File”→“Build Settings”菜单项，打开“Build Settings”窗口，如图9-12所示，然后选择网页格式的游戏平台，接着点击右下角的“Build And Run”按钮编译并运行工程。

待程序编译完成后，系统将自动以网页格式打开游戏，如图9-13所示。在游戏视图中点击“创建本机服务器”按钮即可完成本地服务器的创建。

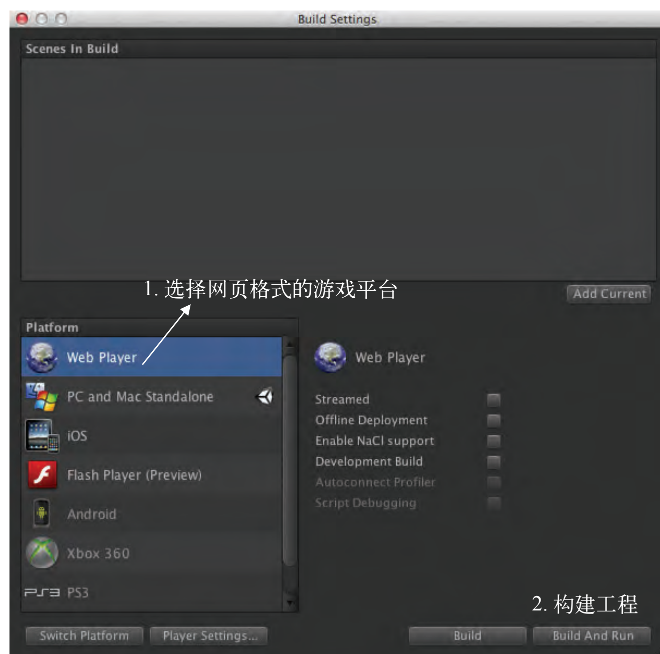


图9-12　构建工程



图9-13　打开工程

9.3.5　客户端连接服务器

本节我们将制作一个客户端去访问刚才创建的服务器。在客户端访问服务器之前，请确保服务器已经开启，并且处于等待连接状态，否则将无法连接，具体代码如代码清单9-8所示。

代码清单9-8　Script_09_08.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_09_08 : MonoBehaviour {

    //服务器的IP地址，这里连接本机地址
    string IP = "127.0.0.1";
    //端口号
    int Port  = 10000;

    void OnGUI()
    {
        //网络连接状态
        switch(Network.peerType)
        {
            //服务器未开启状态
            case NetworkPeerType.Disconnected:
                StartConnect();
            break;
            //成功连接至服务器端
            case NetworkPeerType.Server:

            break;
            //成功连接至客户端
            case NetworkPeerType.Client:

            break;
            //正在尝试连接
            case NetworkPeerType.Connecting:

            break;
        }
    }

    //连接服务器
    void StartConnect()
    {
        if(GUILayout.Button("连接服务器"))
        {
        //客户端开始尝试连接服务器
            NetworkConnectionError error = Network.Connect(IP, Port);
            //如果连接失败，将错误信息打印出来
            Debug.Log("连接状态："+error);
        }
    }
}
```

运行上述代码，可以看到客户端已经正常连接到服务器，此时客户端的相关信息显示在游戏视图中，如图9-14所示。



图9-14　连接成功

9.3.6　实例——多人聊天服务器端

本节将介绍如何构建一个简单的多人聊天服务器端。既然是多人聊天，那么肯定会存在多个客户端，要确保所有的客户端都已连接到服务器，其中任意客户端发出的消息都将通过服务器转发至其他客户端。

为了保证服务器能接受到客户端发来的请求，需要给接收请求脚本所绑定的对象添加网络视图组件。由于服务器脚本绑定在主摄像机中，所以应将网络视图组件添加到摄像机中。添加网络视图组件的方法如下：先选择主摄像机游戏对象，然后在Unity导航菜单栏中选择“Component”→“Miscellaneous”→“Network View”菜单项即可，如图9-15所示。

网络视图组件创建完成后，系统就可以使用RPC来接收网络请求了。值得注意的是，C#语言要求在接收请求方法的上方添加“[RPC]”标识。由于本例使用RequestMessage()方法来接收客户端请求，所以它的上方标示有“[RPC]”字样，本例代码如代码清单9-9所示。

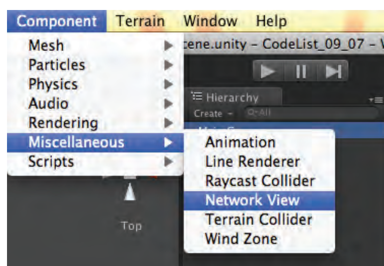


图9-15　添加网络视图组件

代码清单9-9　Script_09_09.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_09_09 : MonoBehaviour {

    //端口号
    int Port  = 10000;
    //聊天信息
    string Message = "";
    //滚动视图位置
    Vector2 scrollPosition;


    void OnGUI()
    {
        //网络连接状态
        switch(Network.peerType)
        {
            //服务器未开启状态
            case NetworkPeerType.Disconnected:
                StartServer();
            break;
            //成功连接至服务器端
            case NetworkPeerType.Server:
                OnServer();
            break;
            //成功连接至客户端
            case NetworkPeerType.Client:
            break;
            //正在尝试连接
            case NetworkPeerType.Connecting:
            break;
        }
    }

    //创建本机服务器
    void StartServer()
    {
```

9

```
        if(GUILayout.Button("创建本机服务器"))
        {
            //创建服务器，允许连接10台主机
            NetworkConnectionError error = Network.InitializeServer(10, Port, false);
            //如果连接失败，将错误信息打印出来
            Debug.Log("连接状态："+error);
        }
    }

    void OnServer()
    {
        GUILayout.Label("服务器创建完毕，等待客户端连接");
        //得到客户端连接的数量
        int length = Network.connections.Length;
        for(int i =0; i<length; i++)
        {
            GUILayout.Label("连接服务器客户端ID" + i);
            GUILayout.Label("连接服务器客户端IP" + Network.connections[i].ipAddress);
            GUILayout.Label("连接服务器客户端端口号" + Network.connections[i].port);
        }
        //断开服务器
        if (GUILayout.Button("断开服务器"))
        {
            Network.Disconnect();
            //重置聊天信息
            Message="";
        }
        //创建一个滚动视图，用来显示聊天信息
        scrollPosition = GUILayout.BeginScrollView(scrollPosition, GUILayout.Width
            (200), GUILayout.Height(500));

        //显示聊天信息
        GUILayout.Box(Message);
        GUILayout.EndScrollView();

    }
    //接收消息
    [RPC]
    void RequestMessage(string message, NetworkMessageInfo info)
    {
        Message +="\n" + "发送者"+ info.sender + ":" + message;
    }
}
```

9.3.7 实例——多人聊天客户端

服务器创建完成后，客户端需要向服务器发送消息请求，并且确保服务器能够将来自某一客户端的请求消息返回给连接到服务器的所有客户端。首先创建一个网络视图组件，然后就可以使用RPC向服务器发送消息了。使用RPC发送消息的方法如下：

```
networkView.RPC("RequestMessage", RPCMode.All, inputMessage);
```

下面简要介绍该方法中各个参数的含义。

- 参数1：接收消息的方法，接收方法上方必须有“[RPC]”标识。
- 参数2：发送模式，可以在这里选择发送对象。
- 参数3：所发送消息的内容。

发送消息的模式共分为5种，具体如下。

- `RPCMode.Server`：向服务器发送。
- `RPCMode.Others`：发送给除发送者外，当前连接到服务器的每一个人。
- `RPCMode.OthersBuffered`：发送给除发送者外，当前连接到服务器的每一个人，并且将数据添加至缓冲区。
- `RPCMode.All`：发送给连接到服务器的所有人，包括发送者和服务器端。
- `RPCMode.AllBuffered`：发送给连接到服务器的所有人，包括发送者和服务器端，并且将数据添加至缓冲区。

在服务器端或客户端中接收消息请求的方法如下。

```
[RPC]
void RequestMessage(string message, NetworkMessageInfo info)
{
    //message：消息内容
    //info：消息附带信息
}
```

至此，读者对服务器与客户端的构建应该已经有了初步的了解。下面将介绍客户端与服务器之间的交互，该过程分为两步，首先建立客户端与服务器之间的连接，然后从客户端向服务器发送消息，如代码清单9-10所示。

代码清单9-10 Script_09_10.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_09_10 : MonoBehaviour {

    //服务器的IP地址，这里连接本机地址
    string IP = "127.0.0.1";
    //端口号
    int Port  = 10000;
    //输入的信息
    string inputMessage = "请输入信息";
    //接收到的信息
    string Message = "";
    //滚动视图位置
    Vector2 scrollPosition;

    void OnGUI()
    {
        //网络连接状态
        switch(Network.peerType)
```

```
    {
        //服务器未开启状态
        case NetworkPeerType.Disconnected:
            StartConnect();
        break;
        //成功连接至服务器
        case NetworkPeerType.Server:

        break;
        //成功连接至客户端
        case NetworkPeerType.Client:
            OnClient();
        break;
        //正在尝试连接
        case NetworkPeerType.Connecting:

        break;
    }
}

//连接服务器
void StartConnect()
{
    if(GUILayout.Button("连接服务器"))
    {
        //客户端开始尝试连接服务器
        NetworkConnectionError error = Network.Connect(IP, Port);
        //如果连接失败，将错误信息打印出来
        Debug.Log("连接状态："+error);
    }
}


void OnClient()
{
    //创建一个滚动视图，用来显示聊天信息
    scrollPosition = GUILayout.BeginScrollView(scrollPosition, GUILayout.Width
        (200), GUILayout.Height(500));
    //显示聊天信息
    GUILayout.Box(Message);
    //创建水平方向视图
    GUILayout.BeginHorizontal();
    //编辑输入内容
    inputMessage = GUILayout.TextArea(inputMessage);
    //发送内容
    if (GUILayout.Button("发送信息"))
    {
        //使用RPC发送内容
        networkView.RPC("RequestMessage", RPCMode.All, inputMessage);
    }
    //结束水平方向视图
    GUILayout.EndHorizontal();

    //断开连接
```

```
            if (GUILayout.Button("断开连接"))
            {
                Network.Disconnect();
                //重置聊天信息
                Message="";
            }
            //结束滚动视图
            GUILayout.EndScrollView();
        }

        //接收消息
        [RPC]
        void RequestMessage(string message, NetworkMessageInfo info)
        {
             Message +="\n" + "发送者"+ info.sender + ":" + message;
        }
    }
```

因为在Unity中不能同时打开两个项目，所以为了便于测试，请读者将工程打包成网页格式后再运行。先打开服务器，然后用客户端连接，确保连接成功后再发送消息。打开客户端1，发送消息“hello!”，而“发送者-1”表示该消息由自身发出，如图9-16所示。然后打开客户端2，发送消息“How are you!”，此时在两个客户端中均可看到彼此发送的内容，如图9-17所示。



图9-16　客户端1

此外，服务器端也正常地接收到了客户端发送的消息。“发送者1”表示第一个连接至服务器的客户端，“发送者2”表示第二个连接至服务器的客户端，如图9-18所示。



图9-17　客户端2

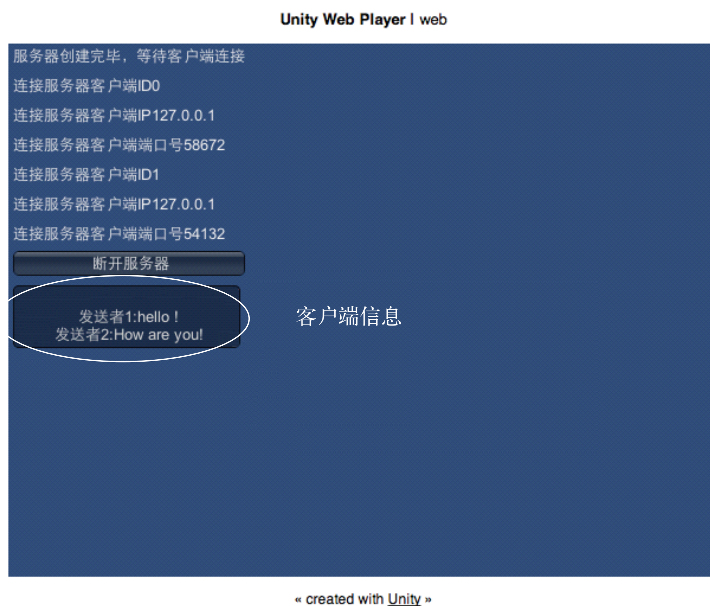


图9-18　服务器端

9.4 游戏实例——简单的网络游戏

网络游戏就是基于客户端与服务器来实现的，所有玩家都处于同一网络中，彼此之间需要互动，比如在自己的游戏画面中可以观察到其他玩家在移动。

在网络游戏中，客户端只负责绘制，而大量的逻辑判断都会在服务器端进行。这样做的好处在于能够有效地防止外挂程序的出现。原因很简单，试想一下，如果把行走逻辑写在客户端中，那么变速外挂就非常容易制作，只需要修改一下移动的数值即可，而如果把它写在服务器端就会安全很多，因为玩家的一切行走逻辑与行走距离都是服务器反馈给客户端的，而外挂是无法修改服务端返回数据的。

本节将制作一款简单的网络游戏，实现角色之间的聊天与移动功能。游戏的操作方式为："W"、"S"键控制角色的前、后移动，"A"、"D"键控制角色的左、右旋转。客户端负责控制角色的移动，并且接收对方发送的聊天内容，当对方移动时，玩家可在自己的屏幕中观察到对方的移动效果，如图9-19所示。

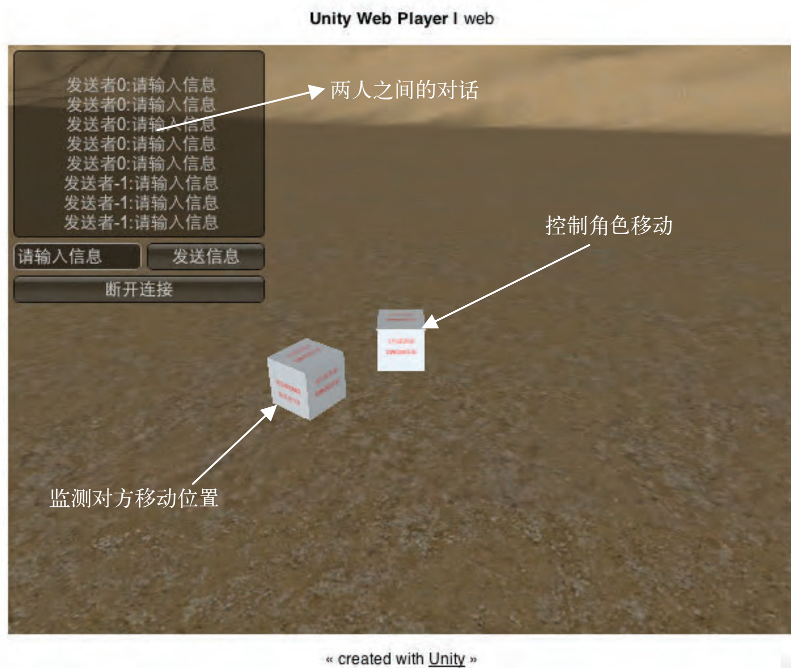


图9-19 客户端

角色移动事件的实现原理是这样的：当玩家按下移动按键时，客户端将按下事件发送至服务器，服务器端判断角色该如何移动才能将事件返回给客户端，当客户端接受到服务器返回的数据后，开始移动角色，同时服务器端还会将其他玩家的移动事件发送给客户端。这样客户端就可以

显示其他玩家的模型位置了。在服务器端中，我们可以清楚地看到客户端的聊天内容以及它们移动模型的事件，如图9-20所示。服务器端代码如代码清单9-11所示。

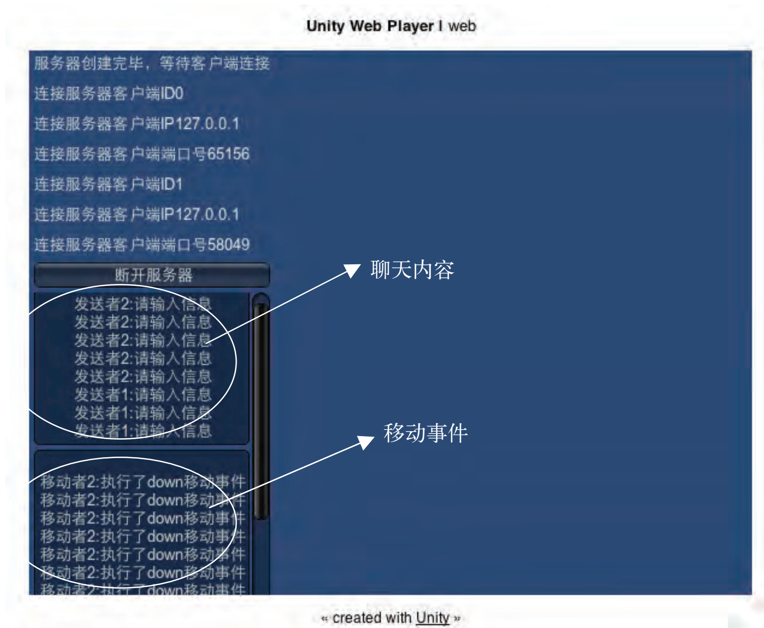


图9-20　服务器端

代码清单9-11　Script_09_11.cs

```
using UnityEngine;
using System.Collections;

public class Script_09_11 : MonoBehaviour {


    //端口号
    int Port  = 10000;
    //聊天信息
    string Message = "";
    //移动信息
    string MoveInfo = "";

    //滚动视图位置
    Vector2 scrollPosition;

    void OnGUI()
    {
        //网络连接状态
        switch(Network.peerType)
        {
```

```
            //服务器未开启状态
            case NetworkPeerType.Disconnected:
                StartServer();
            break;
            //成功连接至服务器端
            case NetworkPeerType.Server:
                OnServer();
            break;
            //成功连接至客户端
            case NetworkPeerType.Client:
            break;
            //正在尝试连接
            case NetworkPeerType.Connecting:
            break;
        }
    }

    //创建本机服务器
    void StartServer()
    {
        if(GUILayout.Button("创建本机服务器"))
        {
            //创建服务器，允许连接10台主机
            NetworkConnectionError error = Network.InitializeServer(10, Port, false);
            //如果连接失败，将错误信息打印出来
            Debug.Log("连接状态："+error);
        }
    }

    void OnServer()
    {
        GUILayout.Label("服务器创建完毕，等待客户端连接");
        //得到客户端连接的数量
        int length = Network.connections.Length;
        for(int i =0; i<length; i++)
        {
            GUILayout.Label("连接服务器客户端ID" + i);
            GUILayout.Label("连接服务器客户端IP" + Network.connections[i].ipAddress);
            GUILayout.Label("连接服务器客户端端口号" + Network.connections[i].port);
        }
        //断开服务器
        if (GUILayout.Button("断开服务器"))
        {
            Network.Disconnect();
            //重置聊天信息
            Message="";
            MoveInfo="";
        }
        //创建一个滚动视图，用来显示聊天信息
        scrollPosition = GUILayout.BeginScrollView(scrollPosition, GUILayout.Width
            (200), GUILayout.Height(Screen.height));

        //显示聊天信息
        GUILayout.Box(Message);
```

```
            //显示玩家移动信息
            GUILayout.Box(MoveInfo);
            GUILayout.EndScrollView();


        }

        //接收消息
        [RPC]
        void RequestMessage(string message, NetworkMessageInfo info)
        {
            Message +="\n" + "发送者"+ info.sender + ":" + message;
        }

        //接收模型移动消息
        [RPC]
        void RequestMove(string message, NetworkMessageInfo info)
        {
            MoveInfo +="\n" + "移动者"+ info.sender + ":执行了" + message + "移动事件";
        }

    }
```

在同时进行游戏的另一个客户端中，我们可以看到第一个角色的位置，而这两个客户端使用的代码是完全一样的，模型在客户端中显示的位置完全取决于服务器，如图9-21所示。

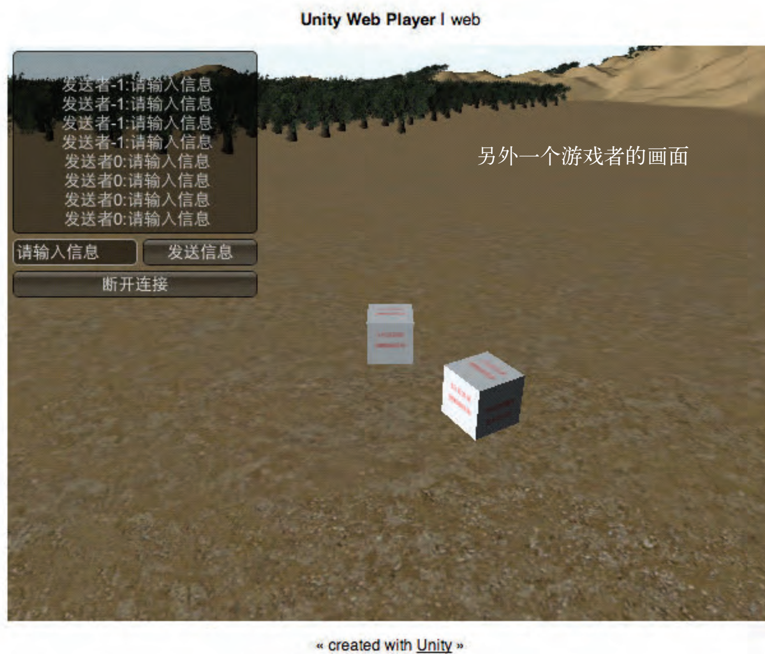


图9-21 另一个客户端

客户端代码如代码清单9-12所示。

代码清单9-12 Script_09_12.cs

```
using UnityEngine;
using System.Collections;

public class Script_09_12 : MonoBehaviour {

    //服务器的IP地址，这里连接本机地址
    string IP = "127.0.0.1";
    //端口号
    int Port  = 10000;
    //输入的信息
    string inputMessage = "请输入信息";
    //接收到的信息
    string Message = "";
    //滚动视图位置
    Vector2 scrollPosition;

    //移动速度
    float speed = 10.0F;
    //旋转速度
    float rotationSpeed = 100.0F;
    //主角控制模型
    GameObject cube0 = null;
    //对方控制模型
    GameObject cube1 = null;

    void Start()
    {
        //得到游戏对象
        cube0 = GameObject.Find("Cube0");
        cube1 = GameObject.Find("Cube1");
    }

    void OnGUI()
    {
        //网络连接状态
        switch(Network.peerType)
        {
            //服务器未开启状态
            case NetworkPeerType.Disconnected:
                StartConnect();
            break;
            //成功连接运行于服务端
            case NetworkPeerType.Server:

            break;
            //成功连接运行于客户端
            case NetworkPeerType.Client:
                OnClient();
            break;
```

```
            //正在尝试连接中
            case NetworkPeerType.Connecting:

            break;
        }

    }

    void FixedUpdate()
    {
        if (Network.isClient)
        {
            float translation = Input.GetAxis("Vertical");
            float rotation = Input.GetAxis("Horizontal");
            if(translation == 1)
            {
                //使用RPC发送向前移动事件
                networkView.RPC("RequestMove", RPCMode.All, "up");
            }
            if(translation == -1)
            {
                //使用RPC发送向后移动事件
                networkView.RPC("RequestMove", RPCMode.All, "down");

            }
            if(rotation == 1)
            {
                //使用RPC发送向左旋转事件
                networkView.RPC("RequestMove", RPCMode.All, "left");

            }
            if(rotation == -1)
            {
                //使用RPC发送向右旋转事件
                networkView.RPC("RequestMove", RPCMode.All, "right");

            }
        }
    }

    //连接服务器
    void StartConnect()
    {
        if(GUILayout.Button("加入游戏"))
        {
            //客户端开始尝试连接服务器
            NetworkConnectionError error = Network.Connect(IP, Port);
            //如果连接失败，将错误信息打印出来
            Debug.Log("连接状态："+error);
        }
    }


    void OnClient()
```

```
{
    //创建一个滚动视图，用来显示聊天信息
    scrollPosition = GUILayout.BeginScrollView(scrollPosition, GUILayout.Width
        (200), GUILayout.Height(500));
    //显示聊天信息
    GUILayout.Box(Message);
    //创建水平方向视图
    GUILayout.BeginHorizontal();
    //编辑输入内容
    inputMessage = GUILayout.TextArea(inputMessage);
    //发送内容
    if (GUILayout.Button("发送信息"))
    {
        //使用RPC发送内容
        networkView.RPC("RequestMessage", RPCMode.All, inputMessage);
    }
    //结束水平方向视图
    GUILayout.EndHorizontal();

    //断开连接
    if (GUILayout.Button("断开连接"))
    {
        Network.Disconnect();
        //重置聊天信息
        Message="";
    }
    //结束滚动视图
    GUILayout.EndScrollView();
}

//接收消息
[RPC]
void RequestMessage(string message, NetworkMessageInfo info)
{
    Message +="\n" + "发送者"+ info.sender + ":" + message;
}


//接收模型移动消息
[RPC]
void RequestMove(string message, NetworkMessageInfo info)
{
    string sender = info.sender.ToString();
    GameObject moveObject = null;

    //自己的移动事件
    if(sender == "-1")
    {
        moveObject = cube0;
    }
    //其他玩家的移动事件
    else
    {
        moveObject = cube1;
```

```
        }

        //根据消息判断事件类型
        int vertical = 0;
        int horizontal = 0;

        if(message == "up")
        {
            //向前移动
            vertical = 1;
        }
        if(message == "down")
        {
            //向后移动
            vertical = -1;
        }
        if(message == "left")
        {
            //向左旋转
            horizontal = 1;
        }
        if(message == "right")
        {
            //向右旋转
            horizontal = -1;
        }
        //移动角色
        float translation = vertical * speed * Time.deltaTime;
        float rotation = horizontal * rotationSpeed * Time.deltaTime;
        moveObject.gameObject.transform.Translate(0, 0, translation);
        moveObject.gameObject.transform.Rotate(0, rotation, 0);

    }

}
```

在上述代码中，我们使用`FixedUpdate()`方法获取用户输入的按键信息，接着使用`networkView.RPC()`方法发送至服务器，而服务器通过`RequestMove()`方法返回角色移动的距离与旋转的角度。

9.5 本章小结

本章主要介绍了Unity中的游戏音频、视频与网络。开始部分讲解了如何在程序中播放音频和视频，其中播放音频需要使用Audio Source组件，而播放视频需要使用电影纹理；接着介绍了游戏中网络方面的相关知识，包括文件的下载、客户端与服务端之间的通信，后者是本章的重点，服务端创建完成后，客户端就可以进行连接，客户端之间的通信通过服务端反馈；最后以一款简单的网络游戏为例，充分向读者诠释了客户端与服务器端之间的通信过程。

第10章

游戏实例——突出重围

本章以一款第一人称射击类游戏为原型，向读者详细介绍了使用Unity开发制作游戏的整个过程，涉及的内容包括游戏界面、主角与敌人逻辑、敌人的AI、胜利与失败条件等。

10.1 游戏状态机

说到游戏状态机，就不得不提到开发中最常用的MVC模型。MVC模型的全称是Model-View-Controller，它将整个游戏开发划分为三大模块：模型组件、视图组件和控制器组件。

模型组件是视图组件与控制器组件之间通信的桥梁，比如，在控制主角移动时，系统先通过控制器输入键盘事件并将具体操作信息发送给模型组件，然后模型组件经过一系列的逻辑计算，得到主角移动后的位置并将相关信息发送给视图组件，最后视图组件接收消息并将主角的位置正确显示在屏幕中。

Unity已经将脚本转化为MVC模式，并且进行了细致的划分。回到Unity项目中，请读者思考这几个常用的方法：`OnGUI()`、`Update()`、`LateUpdate()`和`FixedUpdate()`。以上方法均已被模块化，比如`OnGUI()`方法负责UI的绘制，它就是视图组件；`Update()`、`LateUpdate()`和`FixedUpdate()`负责逻辑更新，属于模型组件；而鼠标、键盘和手柄的事件就是控制器组件。它们相互通信共同完成游戏的主循环。

最后回到本节的主题——游戏状态机中，游戏状态机对游戏进行模块化，并且将其划分为很多不同的游戏状态，在脚本最上层用一个变量来记录当前游戏状态，这样，游戏逻辑与游戏渲染就可以根据当前的游戏状态来执行各自的任务了。因为这个变量凌驾于MVC模式之上，所以在各循环中都可以判断当前的游戏状态。

下面是一段有关游戏状态机的代码：

```
using UnityEngine;
using System.Collections;

public class Script : MonoBehaviour
{

    //游戏状态
    public const int STATE_0 = 0;
    public const int STATE_1 = 1;
    public const int STATE_2 = 2;
    public const int STATE_3 = 3;
```

```
    public const int STATE_4 = 4;

    //记录当前游戏状态
    private int gameState;

    void Start()
    {
        //设置默认游戏状态
        gameState = STATE_0;
    }

    void Update()
    {
        //根据不同状态来更新游戏逻辑
        switch(gameState)
        {
        case STATE_0:

                break;
        case STATE_1:

                break;
        case STATE_2:

                break;
        case STATE_3:

                break;
        case STATE_4:

                break;
        }
    }

    void OnGUI()
    {
        //根据不同状态来绘制游戏
        switch(gameState)
        {
        case STATE_0:

                break;
        case STATE_1:

                break;
        case STATE_2:

                break;
        case STATE_3:

                break;
        case STATE_4:

                break;
        }
    }
}
```

在上述代码中，游戏共分为5个状态，变量`gameState`记录了当前的游戏状态，`Update()`与`OnGUI()`方法用于判断当前的游戏状态并且执行各自的方法。比如，在`Update()`方法中修改当前的游戏状态，紧接着`OnGUI()`方法就会得到修改后的游戏状态，然后根据新的游戏状态来执行绘制方法，这就是经典的游戏状态机制。

10.2　游戏界面

本节主要探讨游戏界面的制作方法，以2D为主。界面包括游戏标题、游戏按钮等控件，它们都是由自定义图片组成的美观的图片资源可有效提升游戏的画面感。

一般情况下，界面由图片视图与按钮视图组成，图片视图只将图片展示在屏幕中，用户无法通过点击触发事件，比如游戏主菜单中的标题。而按钮视图是使用图片制作的按钮，在用户点击按钮后，可执行一些操作，比如点击“开始游戏”按钮，游戏将正式开始。

10.2.1　游戏主菜单

在游戏主菜单中，游戏按钮使用自定义GUI皮肤，而脚本中的贴图与皮肤资源的赋值则以编辑器拖曳的方式完成，如图10-1所示。

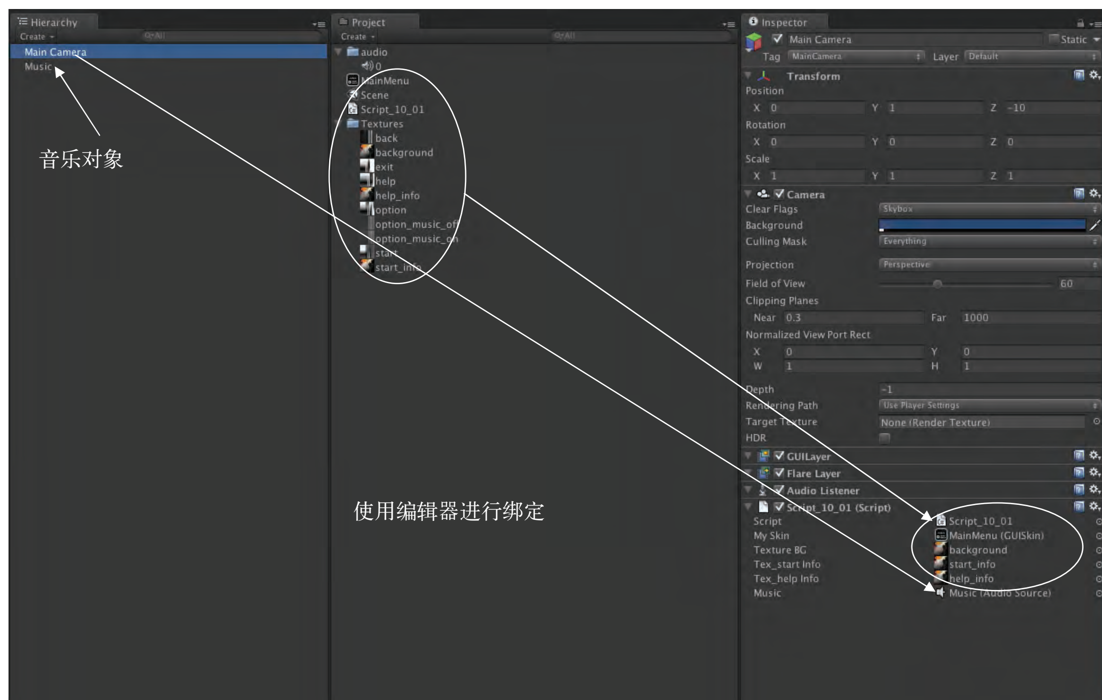


图10-1　绑定数据

游戏主菜单的最终效果如图10-2所示，它包括“开始”、“选项”、“帮助”和“退出”按钮，其制作方法如下，首先将背景图片绘制在屏幕中，然后再将按钮组件添加至此。本例使用游戏状态机将界面划分为不同的状态，选择不同的按钮会执行不同的逻辑，便于后期代码的维护。具体代码如代码清单10-1所示。

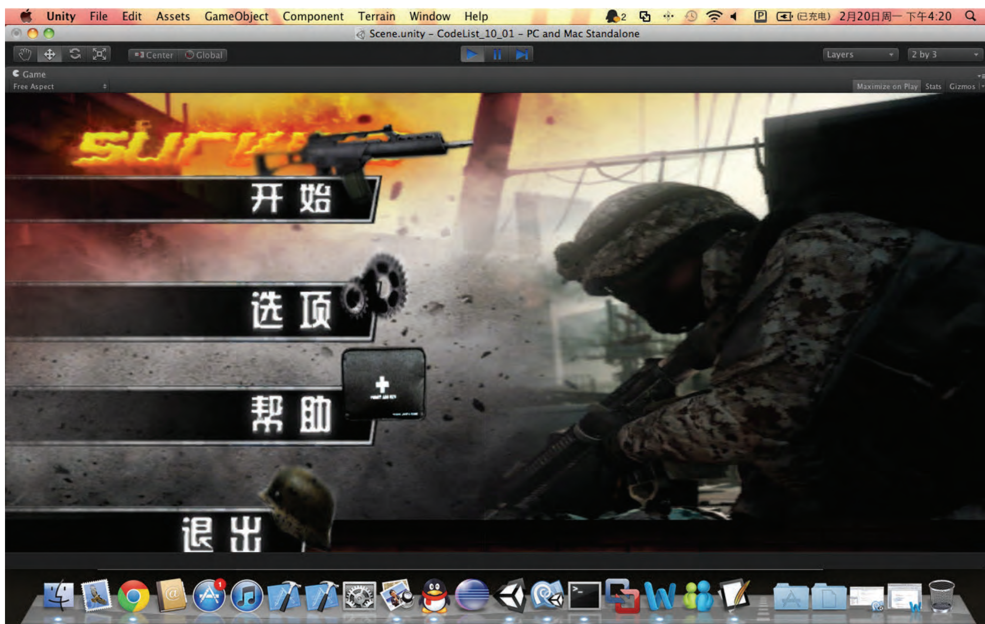


图10-2　游戏界面

代码清单10-1　Script_10_01.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_10_01 : MonoBehaviour
{

    //游戏界面状态机

    //主菜单界面
    public const int STATE_MAINMENU = 0;
    //开始游戏界面
    public const int STATE_STARTGAME = 1;
    //游戏设置界面
    public const int STATE_OPTION = 2;
    //游戏帮助界面
    public const int STATE_HELP = 3;
    //游戏退出界面
    public const int STATE_EXIT = 4;
```

```
//GUI皮肤
public GUISkin mySkin;

//游戏背景贴图
public Texture textureBG;
//开始菜单截图
public Texture tex_startInfo;
//帮助菜单贴图
public Texture tex_helpInfo;

//游戏音乐资源
public AudioSource music;
//当前游戏状态
private int gameState;

void Start()
{
    //初始化游戏状态为主菜单界面
    gameState = STATE_MAINMENU;
}

void OnGUI()
{

    switch(gameState)
    {
        case STATE_MAINMENU:
            //绘制主菜单界面
            RenderMainMenu();
        break;
        case STATE_STARTGAME:
            //绘制游戏开始界面
            RenderStart();
        break;
        case STATE_OPTION:
            //绘制游戏设置界面
            RenderOption();
        break;
        case STATE_HELP:
            //绘制游戏帮助界面
            RenderHelp();
        break;
        case STATE_EXIT:
            //绘制游戏退出界面
            //目前直接关闭并退出游戏
        break;
    }


}
//绘制主菜单界面
void RenderMainMenu()
{
    //设置界面皮肤
    GUI.skin = mySkin;
    //绘制游戏背景图
    GUI.DrawTexture(new Rect(0,0,Screen.width,Screen.height),textureBG);
```

```
        //开始游戏按钮
        if(GUI.Button(new Rect (0,30,623,153),"","start"))
        {
            //进入开始游戏状态
            //由于目前处于测试阶段
            //后期会在这里重新载入新的游戏场景
            gameState = STATE_STARTGAME;
        }
        //游戏设置按钮
        if(GUI.Button(new Rect (0,180,623,153),"","option"))
        {
            //进入开始游戏状态
            gameState = STATE_OPTION;
        }
        //游戏帮助按钮
        if(GUI.Button(new Rect (0,320,623,153),"","help"))
        {
            //进入游戏帮助状态
            gameState = STATE_HELP;
        }
        //游戏退出按钮
        if(GUI.Button(new Rect (0,470,623,153),"","exit"))
        {
            //退出游戏
            Application.Quit();
        }
    }
    //绘制游戏开始界面
    void RenderStart()
    {
        GUI.skin = mySkin;
        GUI.DrawTexture(new Rect(0,0,Screen.width,Screen.height),tex_startInfo);
        //绘制返回按钮
        if(GUI.Button(new Rect (0,500,403,78),"","back"))
        {
            //返回游戏主菜单
            gameState = STATE_MAINMENU;
        }
    }
    //绘制游戏帮助界面
    void RenderHelp()
    {
        GUI.skin = mySkin;
        GUI.DrawTexture(new Rect(0,0,Screen.width,Screen.height),tex_helpInfo);
        if(GUI.Button(new Rect (0,500,403,78),"","back"))
        {
            gameState = STATE_MAINMENU;
        }
    }
    //绘制游戏设置界面
    void RenderOption()
    {
        GUI.skin = mySkin;
        GUI.DrawTexture(new Rect(0,0,Screen.width,Screen.height),textureBG);

        //开启音乐按钮
        if(GUI.Button(new Rect (0,0,403,75),"","music_on"))
```

```
            {
                if (!music.isPlaying)
                {
                    //播放音乐
                    music.Play();
                }

            }
            //关闭音乐按钮
            if(GUI.Button(new Rect (0,200,403,75),"","music_off"))
            {
                //关闭音乐
                music.Stop();
            }
            //返回按钮
            if(GUI.Button(new Rect (0,500,403,78),"","back"))
            {
                //返回游戏主菜单
                gameState = STATE_MAINMENU;
            }
        }
    }
```

本例的游戏状态机包括游戏主菜单状态、开始游戏状态、游戏选项状态和游戏帮助状态，系统使用整型变量gameState来记录当前的游戏状态。进入不同的游戏界面将修改当前gameState记录的游戏状态，系统会在OnGUI()方法中更新游戏界面的绘制逻辑。

10.2.2 制作角色血条

角色血条是游戏中较为常见的元素，主要用于记录我方或敌方角色的当前生命值。通常，角色血条以红色做面、黑色做底，角色受到伤害或者增加生命值时红色的面会随之缩放，从而实现减血与加血。本例制作的角色血条如图10-3所示，具体代码如代码清单10-2所示。

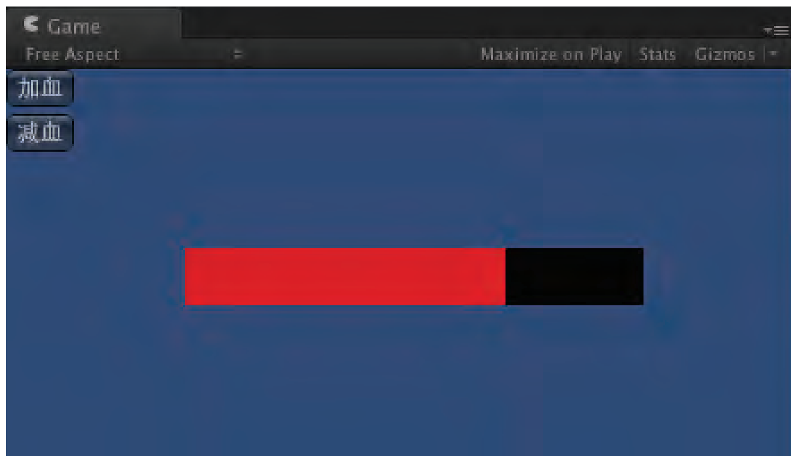


图10-3 游戏血条

代码清单10-2　Script_10_02.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_10_02 : MonoBehaviour {


    //红色血条
    public Texture2D blood_red;
    //黑色血条
    public Texture2D blood_black;
    //当前生命值
    private int HP = 100;

    void OnGUI()
    {
        if(GUILayout.RepeatButton("加血"))
        {
            //增加生命值
            if(HP<100)
            {
                HP++;
            }
        }
        if(GUILayout.RepeatButton("减血"))
        {
            //减少生命值
            if(HP>0)
            {
                HP--;
            }
        }
        //根据当前生命值计算红色血条显示的宽度
        int blood_width = blood_red.width * HP/100;
        //绘制黑色血条
        GUI.DrawTexture(new Rect(100,100,blood_black.width,blood_black.height),
            blood_black);
        //绘制红色血条
        GUI.DrawTexture(new Rect(100,100,blood_width,blood_red.height),blood_red);
    }
}
```

本例使用整型变量HP记录当前生命值，满血状态下该值为100。在屏幕中点击“加血”按钮将增加该数值，点击“减血”按钮将减少该数值，然后程序根据HP的数值即可计算出红色血条矩形的整体显示区域。

10.2.3　制作图片数字

由于使用程序绘制的数字局限性较强，所以通常会使用图片来绘制数字。这样绘制的数字比较美观，它的制作原理如下：首先取得需要显示数字的个位、十位、百位……的数组序列，接着

在图片数组中寻找对应的图片，然后依次将它们绘制在屏幕中即可。为了便于读取图片资源，我们将数字图片资源放置在Textures文件夹中，如图10-4所示。本例代码如代码清单10-3所示。



图10-4　图片数字

代码清单10-3　Script_10_03.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_10_03 : MonoBehaviour {

    //存储图片资源数组
    Object[] texmube;
    //测试整型数据
    int mumber = 1980;

    void Start()
    {
        //读取图片资源
        texmube = Resources.LoadAll("Textures");
    }
    void OnGUI()
    {
        //绘制图片数字
        DrawImageNumber(0,100,mumber,texmube);
    }

    /**
    * int x: 绘制数字，x轴坐标
    * int y: 绘制数字，y轴坐标
    * Object[] texmube 绘制的图片数组资源
    */
    void DrawImageNumber(int x,int y,int mumber,Object[] texmube)
    {
        //将整型数据转换成字符数组
        char[] chars = mumber.ToString().ToCharArray();
```

```
            //计算图片的宽度与高度
            Texture2D tex = (Texture2D)texmube[0];
            int width = tex.width;
            int height = tex.height;
            //遍历字符数组
            foreach (char s in chars)
            {
                //得到整型数据的每一位
                int i = int.Parse(s.ToString()) ;
                //绘制图片数字
                GUI.DrawTexture(new Rect(x,0,width,height),(Texture2D)texmube[i]);
                x += width;
            }
        }

    }
```

本例封装了一个绘制图片数字的方法`DrawImageNumber()`，其实现原理是，首先将需要绘制的数字强转为字符数组，然后使用循环遍历得到数字的个位、十位、百位……，最后通过位数的ID找到对应的图片资源，并使用`DrawTexture()`方法将图片数字绘制在屏幕中。运行上述代码，获得的效果如图10-5所示。

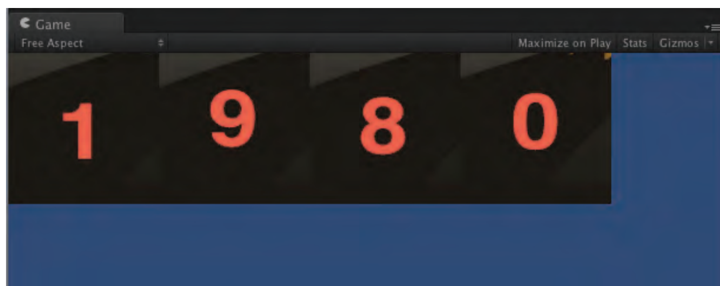


图10-5　图片数字

10.3　游戏逻辑

在游戏中，大量的判断都是通过游戏逻辑来完成的，比如角色的移动、发射子弹与击打敌人的动作、敌人中弹后的效果、角色减血等。本游戏为第一人称射击类游戏，因此可以使用角色控制器组件来完成第一人称视角的制作。

10.3.1　发射子弹与击打目标

首先将角色控制器组件添加至游戏视图，然后将敌人的模型对象也添加至此。由于本游戏是第一人称视角游戏，所以需要对游戏视图添加枪口的准心，这里将准心绑定在鼠标上，它会跟随鼠标移动。接下来需要监听鼠标的点击事件，当玩家瞄准敌人模型对象并且按下鼠标左键时，敌人中枪。然后在屏幕左上角制作敌人的血条，当玩家瞄准敌人模型对象时，系统开始绘制血条，

击中目标后血量开始减少，直到为0时敌人死亡。敌人死亡后立即销毁敌人对象并将其从游戏视图中彻底删除。

为了让玩家感受到击中目标后的回馈，本例制作了震动效果，效果如图10-6所示。当玩家击中目标时，角色瞬间向后移动一段距离，待敌人减血完毕后再恢复角色的初始位置。本例中，控制主角的逻辑脚本代码如代码清单10-4所示。



图10-6　击打目标

代码清单10-4　MouseLook.cs文件

```
using UnityEngine;
using System.Collections;

[AddComponentMenu("Camera-Control/Mouse Look")]
public class MouseLook : MonoBehaviour
{
    //枚举角色鼠标位置
    public enum RotationAxes { MouseXAndY = 0, MouseX = 1, MouseY = 2 }
    public RotationAxes axes = RotationAxes.MouseXAndY;
    float rotationY = 0F;
    //鼠标准心
    public Texture2D tex_fire;


    void Start ()
    {
        //取消默认鼠标图标
        Screen.showCursor = false;
```

```
        if (rigidbody)
            rigidbody.freezeRotation = true;
    }

    void Update ()
    {
        //计算摄像机的旋转
        if (axes == RotationAxes.MouseX)
        {
            //旋转角色
            transform.Rotate(0, Input.GetAxis("Mouse X"), 0);
        }
        else
        {
            //设置角色欧拉角
            transform.localEulerAngles = new Vector3(-rotationY,
                transform.localEulerAngles.y, 0);
        }

    }

    void OnGUI()
    {
        if(tex_fire)
        {
            //绘制鼠标准心
            float x = Input.mousePosition.x - (tex_fire.width>>1);
            float y = Input.mousePosition.y + (tex_fire.height>>1);
            GUI.DrawTexture(new Rect(x,Screen.height - y,tex_fire.width,tex_fire.
                height),tex_fire);
        }

    }
}
```

敌人的逻辑脚本代码如代码清单10-5所示。

代码清单10-5　Script_10_04.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_10_04 : MonoBehaviour
{

    //是否绘制敌人血条
    bool showBlood = false;

    //血条资源贴图
    public Texture2D tex_red;
    public Texture2D tex_black;
    //生命值
    private int HP = 100;
    //主角对象
    private GameObject hero;
```

```
void Start()
{
    //获取主角对象
    hero = GameObject.Find("/Hero");
}

void Update()
{
     //始终朝向主角
     transform.LookAt(hero.transform);
}

void OnGUI()
{
    if(showBlood)
    {
        //绘制敌人血条
        int blood_width = tex_red.width * HP/100;
        GUI.DrawTexture(new Rect(5,5,tex_black.width,tex_black.height),
            tex_black);
        GUI.DrawTexture(new Rect(5,5,blood_width,tex_red.height),tex_red);


    }
}

void OnMouseDown()
{
    //击中敌人目标
    if(HP > 0)
    {
        //减血
        HP-=5;
        //击打回馈
        transform.Translate(Vector3.back * 0.1F);
    }else
    {
        //敌人死亡
        Destroy (gameObject);
    }
}

void OnMouseUp()
{
    //击打回馈还原
    transform.Translate(Vector3.forward * 0.1F);
}

void OnMouseOver()
{
    //开始绘制血条
    showBlood = true;
}

void OnMouseExit()
```

```
        {
            //结束绘制血条
            showBlood = false;
        }

    }
```

本例共使用了两个脚本，MouseLook.cs脚本用于监测鼠标事件，并实时获取鼠标的当前位置，从而计算出主角的当前朝向。Script_10_04.cs脚本用于更新敌人逻辑，当鼠标准心对准敌人时就显示敌人的血条。按下鼠标左键即可向敌人开火，敌人每次中弹将减少5点血量，当血量少于0时敌人死亡，其游戏对象被销毁。

本例增加了敌人自动朝向主角的功能，通过`Update()`方法中的`LookAt()`方法实现，其参数为角色的朝向，这里将主角的位置`hero.transform`传入该方法，这就意味着，无论主角怎样移动，敌人都将通过自身旋转始终面朝主角。

10.3.2　敌人的AI

AI即人工智能，一款游戏的可玩性基本上是由敌人的AI来决定的。通常，AI会被添加给游戏中非主角控制的角色，比如敌人。本例将AI添加给敌人角色，敌人AI脚本代码如代码清单10-6所示。默认情况下，敌人处于巡逻状态。他们每隔两秒思考一次，以决定自己下一步的动作——旋转一个角度或继续向前行走进行巡逻。另外，如果敌人在巡逻时发现主角，就会进入攻击状态，并且向主角所在的位置奔跑。图10-7为正在巡逻的敌人。



图10-7　敌人巡逻

代码清单10-6 Script_10_05.cs文件

```
using UnityEngine;
using System.Collections;

public class Script_10_05 : MonoBehaviour
{
    //敌人状态

    //敌人站立状态
    public const int STATE_STAND = 0;
    //敌人行走状态
    public const int STATE_WALK = 1;
    //敌人奔跑状态
    public const int STATE_RUN = 2;
    //记录敌人的当前状态
    private int enemyState;
    //主角对象
    private GameObject hero;
    //备份上一次的敌人思考时间
    private float backUptime;
    //敌人思考下一次行为的时间
    public const int AI_THINK_TIME = 2;
    //敌人的巡逻范围
    public const int AI_ATTACK_DISTANCE = 10;

    void Start()
    {
        //得到主角对象
        hero = GameObject.Find("/Hero");
        //设置敌人的默认状态为站立
        enemyState = STATE_STAND;
    }


    void Update()
    {
        //判断敌人与主角的距离
        if(Vector3.Distance(transform.position,hero.transform.position) <
           AI_ATTACK_DISTANCE)
        {
             //敌人进入奔跑状态
             gameObject.animation.Play("run");
             enemyState = STATE_RUN;
             //设置敌人面朝主角方向
             transform.LookAt(hero.transform);
        }
        //敌人进入巡逻状态
        else
        {
            //计算敌人的思考时间
            if(Time.time - backUptime >=AI_THINK_TIME)
            {
                //敌人开始思考
```

```
                    backUptime = Time.time;

                    //取得0～2之间的随机数
                    int rand = Random.Range(0,2);

                    if(rand == 0)
                    {
                        //敌人进入站立状态
                        gameObject.animation.Play("idle");
                        enemyState = STATE_STAND;
                    }
                    else if(rand == 1)
                    {
                        //敌人进入行走状态
                        //敌人随机旋转角度
                        Quaternion rotate = Quaternion.Euler(0,Random.Range(1,5) *90,0);
                        //1秒内完成敌人旋转
                        transform.rotation = Quaternion.Slerp(transform.rotation,
                            rotate, Time.deltaTime*1000);
                        //播放行走动画
                        gameObject.animation.Play("walk");
                        enemyState = STATE_WALK;
                    }
                }
            }

                switch(enemyState)
                {
                case STATE_STAND:

                    break;
                case STATE_WALK:
                    //敌人行走
                    transform.Translate(Vector3.forward *Time.deltaTime);
                    break;
                case STATE_RUN:
                    //敌人朝向主角奔跑
                    if(Vector3.Distance(transform.position,hero.transform.position) >3)
                    {
                        transform.Translate(Vector3.forward *Time.deltaTime * 3);
                    }
                    break;
                }
        }
    }
```

本例使用`Update()`方法中的`Distance()`方法来计算当前主角与敌人之间的距离，该方法的第一个参数表示敌人的位置，第二个参数表示主角的位置，该方法的返回值即为它们之间的距离。如果主角与敌人之间的距离大于10米，敌人将每2秒进行一次思考。此时使用`Random.Range(0,2)`方法来获取一个0～2之间的随机数字，当随机数为0时，系统将播放敌人的站立动画，而当随机数为1时，敌人将随机旋转一个角度并向前移动一段距离，系统则播放其行走动画，直到敌人下一次思考。当主角靠近敌人时，敌人面朝主角并且进入奔跑状态，如图10-8所示。



图10-8 敌人奔跑

10.3.3 增加敌人预设

下面使用预设在游戏世界中创建更多的敌人对象，本例共添加了10个敌人，如图10-9所示。

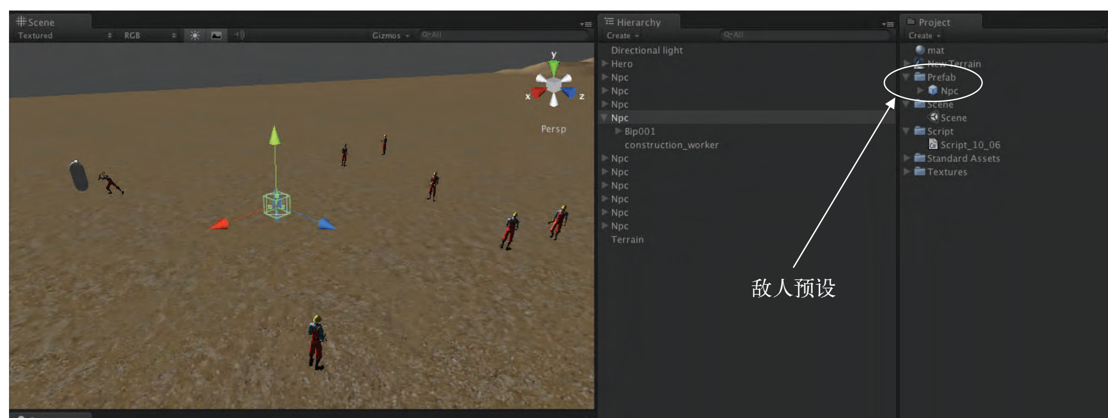


图10-9 敌人预设

此时运行游戏，可看到如图10-10所示的界面，其中的每个敌人对象都在有条不紊地完成着各自的游戏周期。



图10-10　敌人行动

10.4　完整的游戏

本节将结合之前讲解的知识点完成这款游戏的制作。图10-11所示为游戏中用到的所有资源，箱子预设用来创建若干加血箱，主角触碰到这些箱子即可刻回复50点血量；敌人预设用来创建敌人对象，敌人的一切行为都由脚本Script_Enemy.cs控制；主角对象用来控制玩家角色的移动与攻击，控制主角的脚本为Script_Game.cs。

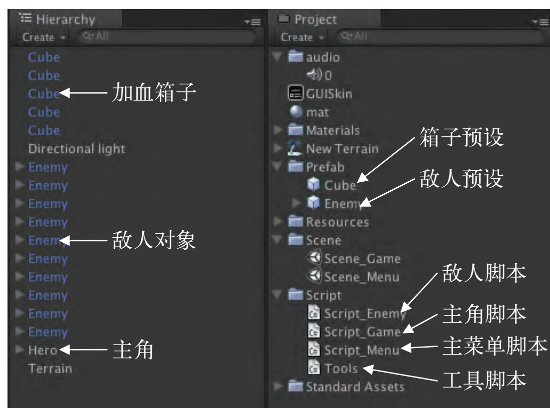


图10-11　游戏资源

本游戏共包含两个场景，第一个场景为主菜单场景，第二个场景为真实游戏场景。运行游戏后，系统首先进入主菜单场景，如图10-12所示，在游戏主菜单中单击“开始”按钮，游戏将切换至真实游戏场景。另外，主菜单场景中的“选项”按钮可控制音乐的开、关，“帮助”按钮用于显示游戏的操作说明与相关信息，“退出”按钮用于结束游戏。



图10-12　游戏主菜单

点击“选项”按钮，进入游戏设置菜单，如图10-13所示，可以在此设置游戏音乐的开、关，设置完成后，点击“返回”按钮继续游戏。



图10-13　游戏选项

单击“帮助”按钮，进入游戏帮助界面，如图10-14所示，这里简要记录了一些游戏的操作方法与相关信息。



图10-14　游戏帮助

下面介绍游戏场景中的内容。为了加强游戏的可玩性，我们在游戏中添加了一类特殊的游戏对象——加血箱。玩家可以控制主角触碰加血箱获得“回血”效果。加血箱只能使用一次，使用后即刻消失。

游戏界面的左上角是玩家当前所瞄准的目标敌人的血条，左下角为主角当前的血条，这里使用图片数字来记录敌人与主角的生命值，如图10-15所示。



图10-15　开始游戏

游戏的胜利条件是将敌人全部杀死，如图10-16所示。

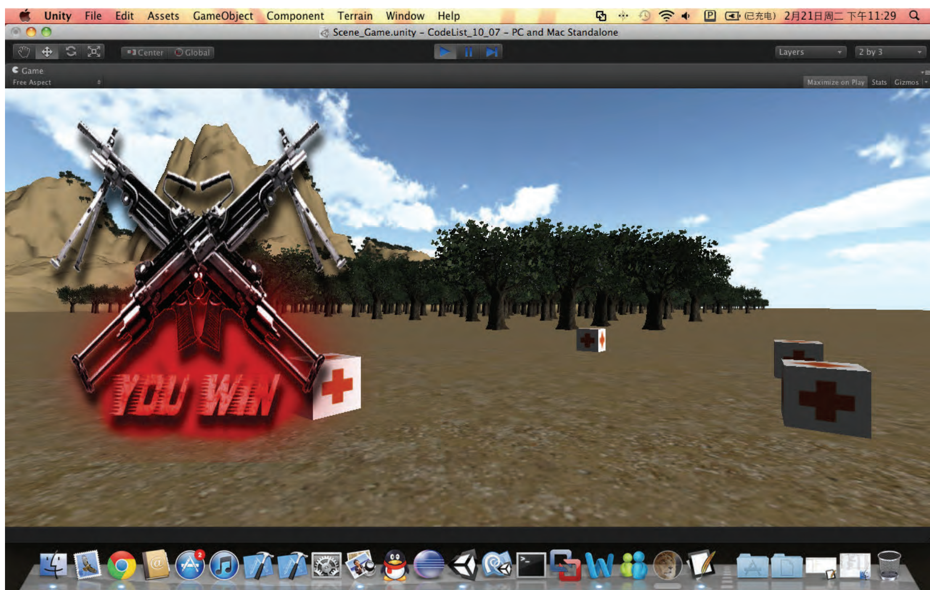


图10-16 游戏胜利

当主角的生命值为0时，游戏失败，如图10-17所示。

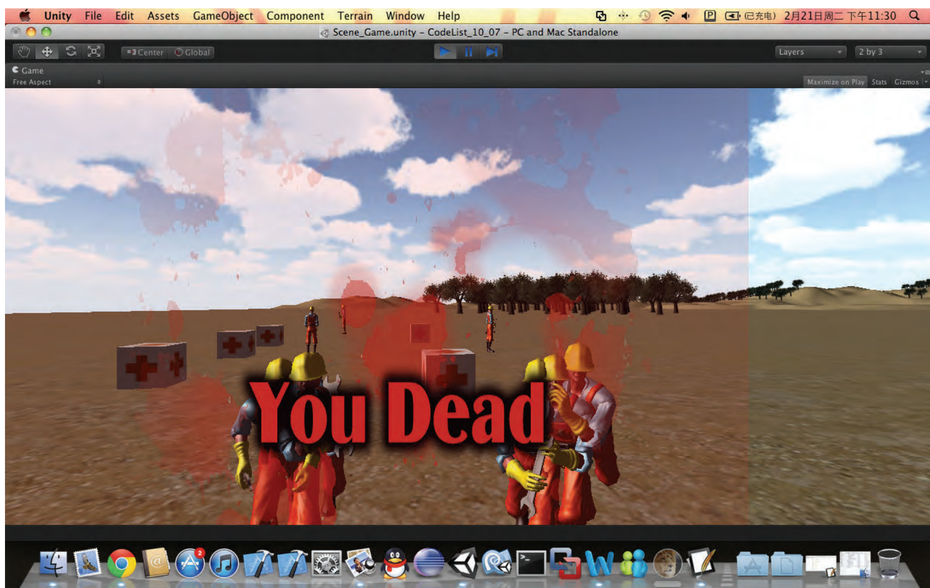


图10-17 游戏失败

下面将介绍在完整的游戏中所用到的代码。处理敌人逻辑的代码如代码清单10-7所示，该脚本绑定在敌人对象中。

代码清单10-7　Script_Enemy.cs

```
using UnityEngine;
using System.Collections;

public class Script_Enemy : MonoBehaviour
{
    //敌人状态

    //敌人站立状态
    public const int STATE_STAND = 0;
    //敌人行走状态
    public const int STATE_WALK = 1;
    //敌人奔跑状态
    public const int STATE_RUN = 2;
    //敌人暂停状态
    public const int STATE_PAUSE = 3;

    //记录敌人的当前状态
    private int enemyState;
    //主角对象
    private GameObject hero;

    //备份上一次敌人的思考时间
    private float backUptime;
    //敌人思考下一次行为的时间
    public const int AI_THINK_TIME = 2;
    //敌人的巡逻范围
    public const int AI_ATTACK_DISTANCE = 10;

    //是否绘制敌人血条
    bool showBlood = false;

    //血条资源贴图
    public Texture2D tex_red;
    public Texture2D tex_black;

    //生命值贴图
    public Texture2D tex_hp;

    //生命值
    private int HP = 100;

    //敌人仇恨
    private bool ishatred = false;

    //图片数字资源
    Object[] texmube;

    void Start()
```

```
{
    //得到主角对象
    hero = GameObject.Find("/Hero");
    //读取图片资源
    texmube = Resources.LoadAll("number");
    //设置敌人的默认状态为站立
    enemyState = STATE_STAND;

}

void OnGUI()
{
    if(showBlood)
    {
        //绘制生命值贴图
        GUI.DrawTexture(new Rect(5,0,tex_hp.width,tex_hp.height),tex_hp);
        //绘制主角生命值
        Tools.DrawImageNumber(200,0,HP,texmube);

        //绘制敌人血条
        int blood_width = tex_red.width * HP/100;
        GUI.DrawTexture(new Rect(5,50,tex_black.width,tex_black.height),
            tex_black);
        GUI.DrawTexture(new Rect(5,50,blood_width,tex_red.height),tex_red);


    }
}
void Update()
{
    //判断敌人与主角的距离
    if(Vector3.Distance(transform.position,hero.transform.position) <
        AI_ATTACK_DISTANCE || ishatred)
    {
         //敌人进入奔跑状态
         gameObject.animation.Play("run");
         enemyState = STATE_RUN;
         //设置敌人面朝主角方向
         transform.LookAt(hero.transform);
         //随机攻击主角
         int rand = Random.Range(0,20);
         if(rand == 0)
         {
            hero.SendMessage("HeroHurt");
         }
    }
    //敌人进入巡逻状态
    else
    {
        //计算敌人的思考时间
        if(Time.time - backUptime >=AI_THINK_TIME)
        {
            //敌人开始思考
            backUptime = Time.time;
```

```
                //取得0～2之间的随机数
                int rand = Random.Range(0,2);

                if(rand == 0)
                {
                    //敌人进入站立状态
                    gameObject.animation.Play("idle");
                    enemyState = STATE_STAND;
                }
                else if(rand == 1)
                {
                    //敌人进入行走状态
                    //敌人随机旋转角度
                    Quaternion rotate = Quaternion.Euler(0,Random.Range(1,5) *90,0);
                    //1秒内完成敌人旋转
                    transform.rotation = Quaternion.Slerp(transform.rotation,
                        rotate, Time.deltaTime*1000);
                    //播放行走动画
                    gameObject.animation.Play("walk");
                    enemyState = STATE_WALK;
                }
            }
        }

            switch(enemyState)
            {
            case STATE_STAND:

                break;
            case STATE_WALK:
                //敌人行走
                transform.Translate(Vector3.forward *Time.deltaTime);
                break;
            case STATE_RUN:
                //敌人朝向主角奔跑
                if(Vector3.Distance(transform.position,hero.transform.position) >3)
                {
                    transform.Translate(Vector3.forward *Time.deltaTime * 3);
                }
                break;

            }
    }

    void OnMouseDown()
    {
        //击中敌人目标
        if(HP > 0)
        {
            //减血
            HP-=5;
            //击打回馈
            transform.Translate(Vector3.back * 0.1F);
```

```
                //击中敌人后立刻进入战斗状态
                ishatred = true;
            }else
            {
                //敌人死亡
                Destroy (gameObject);
                //发送死亡消息
                hero.SendMessage("EnemyHurt");
            }
        }

        void OnMouseUp()
        {
            //击打回馈还原
            transform.Translate(Vector3.forward * 0.1F);
        }

        void OnMouseOver()
        {
            //开始绘制血条
            showBlood = true;
        }

        void OnMouseExit()
        {
            //结束绘制血条
            showBlood = false;
        }

    }
```

在脚本中敌人的状态分为4种：站立状态、行走状态、奔跑状态和暂停状态，代码中的enemyState整型变量记录了敌人的当前状态。敌人未发现主角时，每隔两秒思考一次，之后可能进入站立状态，也可能进入行走状态。而敌人发现主角后，将进入奔跑状态，跑向主角，并进行随机攻击。此时，hero.SendMessage("HeroHurt")方法将向主角对象发送攻击消息，然后在主角对象的脚本中实现HeroHurt方法，进行主角减血。

主角对象的脚本绑定在第一人称摄像机中，用于处理游戏逻辑，其代码如代码清单10-8所示。

代码清单10-8 Script_Game.cs

```
using UnityEngine;
using System.Collections;

public class Script_Game : MonoBehaviour
{
    //游戏状态机

    //游戏中状态
    public const int STATE_GAME =0;
    //游戏胜利状态
    public const int STATE_WIN =1;
    //游戏失败状态
```

```
public const int STATE_LOSE =2;

//枚举角色鼠标位置
public enum RotationAxes { MouseXAndY = 0, MouseX = 1, MouseY = 2 }
public RotationAxes axes = RotationAxes.MouseXAndY;
float rotationY = 0F;
//鼠标准心
public Texture2D tex_fire;

//血条资源贴图
public Texture2D tex_red;
public Texture2D tex_black;

//生命值贴图
public Texture2D tex_hp;

//战斗胜利资源贴图
public Texture2D tex_win;
//战斗失败资源贴图
public Texture2D tex_lose;
//游戏音乐资源
public AudioSource music;

//主角生命值
public int HP = 100;

//图片数字资源
Object[] texmube;


//当前游戏状态
int gameState;


void Start()
{
    //取消默认鼠标图标
    Screen.showCursor = false;
    //读取图片资源
    texmube = Resources.LoadAll("number");
    //设置默认状态为游戏中
    gameState = STATE_GAME;

    if (rigidbody)
        rigidbody.freezeRotation = true;
}


void Update()
{

    switch(gameState)
    {
    case STATE_GAME:
```

```
            UpdateGame();
            break;
        case STATE_WIN:
        case STATE_LOSE:
            if (Input.GetKey (KeyCode.Escape))
            {
                Application.LoadLevel ("Scene_Menu");
            }
            break;
        }

    }

    void OnGUI()
    {

        switch(gameState)
        {
        case STATE_GAME:
            RenderGame();
            break;
        case STATE_WIN:
            GUI.DrawTexture(new Rect(0,0,tex_win.width,tex_win.height),tex_win);
            break;
        case STATE_LOSE:
            GUI.DrawTexture(new Rect(0,0,tex_lose.width,tex_lose.height),tex_lose);
            break;
        }
    }


    void UpdateGame()
    {
        if (Input.GetKey (KeyCode.Escape))
        {
            Application.LoadLevel ("Scene_Menu");
        }

        //计算摄像机的旋转
        if (axes == RotationAxes.MouseX)
        {
            //旋转角色
            transform.Rotate(0, Input.GetAxis("Mouse X"), 0);
        }
        else
        {
            //设置角色欧拉角
            transform.localEulerAngles = new Vector3(-rotationY,
                transform.localEulerAngles.y, 0);
        }
    }

    void RenderGame()
    {
```

```
    if(tex_fire)
    {
        //绘制鼠标准心
        float x = Input.mousePosition.x - (tex_fire.width>>1);
        float y = Input.mousePosition.y + (tex_fire.height>>1);
        GUI.DrawTexture(new Rect(x,Screen.height - y,tex_fire.width,tex_fire.
            height),tex_fire);
    }

        //绘制主角血条
        int blood_width = tex_red.width * HP/100;
        GUI.DrawTexture(new Rect(5,Screen.height -50,tex_black.width,tex_black.
            height),tex_black);
        GUI.DrawTexture(new Rect(5,Screen.height - 50,blood_width,tex_red.
            height),tex_red);

        //绘制生命值贴图
        GUI.DrawTexture(new Rect(5,Screen.height - 80,tex_hp.width,tex_hp.
            height),tex_hp);
        //绘制主角生命值
        Tools.DrawImageNumber(200,Screen.height - 80,HP,texmube);
}

//主角被攻击
void HeroHurt()
{
    HP--;
    if(HP <=0)
    {
        HP = 0;
        gameState = STATE_LOSE;
    }
}
//触碰道具加血
void HeroAddBlood()
{
    HP+=50;
    if(HP>=100)
    {
        HP = 100;
    }
}

//敌人被攻击
void EnemyHurt()
{
    GameObject []enemy = GameObject.FindGameObjectsWithTag("enemy");
    //敌人对象数字长度为1表示敌人全部死亡
    if(enemy.Length == 1)
    {
        gameState = STATE_WIN;
    }
}
```

```
        void OnControllerColliderHit(ControllerColliderHit hit)
        {
            //获取角色控制器碰撞到的游戏对象
            GameObject collObj = hit.gameObject;
            //是否为加血箱对象
            if(collObj.name == "Cube")
            {
                //主角加血
                HeroAddBlood();
                //销毁箱子
                Destroy(collObj);
            }

        }
    }
```

本脚本主要更新游戏的主逻辑，游戏状态可分为3种，即游戏中状态、游戏胜利状态和游戏失败状态。整型变量gameState记录了游戏的当前状态，通过OnGUI()方法与Update()方法渲染并更新当前的游戏状态。主角被攻击时，系统调用HeroHurt()方法减少主角1点生命值，直到其生命值小于0，游戏进入失败状态。主角攻击敌人时，系统调用EnemyHurt()方法，并使用FindGameObjectsWithTag()方法获取当前未死亡的敌人对象数组，然后根据数组的长度判断敌人的数量，如果为0，则表示敌人全部死亡，游戏进入胜利状态。当主角触碰到加血箱子时，系统调用HeroAddBlood()方法为主角进行加血。

绘制图片数字的工具类代码如代码清单10-9所示。

代码清单10-9 Tools.cs

```
    using UnityEngine;
    using System.Collections;

    public class Tools : MonoBehaviour {

       /**
        * int x: 绘制数字,x轴坐标
        * int y: 绘制数字,y轴坐标
        * Object[] texmube 绘制的图片数组资源
        */
        public static void DrawImageNumber(int x,int y,int mumber,Object[] texmube)
        {
            //将整型数据转换成字符数组
            char[] chars = mumber.ToString().ToCharArray();
            //计算图片的宽度与高度
            Texture2D tex = (Texture2D)texmube[0];
            int width = tex.width;
            int height = tex.height;
            //遍历字符数组
            foreach (char s in chars)
            {
                //得到每一位整型数据
                int i = int.Parse(s.ToString()) ;
                //绘制图片数字
```

```
                GUI.DrawTexture(new Rect(x,y,width,height),(Texture2D)texmube[i]);
                x += width;
            }
        }
    }
```

工具类主要用于存放静态方法，其他类无需实例化其对象，直接通过其类名即可调用类中的方法。因为更新主角与更新敌人的脚本都会用到绘制图片数字的方法，所以我们将此方法封装在工具类当中，方便不同的类调用。

10.5　本章小结

本章通过一个游戏实例向读者介绍了Unity 3D的详细开发过程。游戏运行后，系统将首先打开游戏主菜单场景，其中存放着一些功能性的按钮，如“开始”、“选项”等。单击“开始”按钮，系统将进入游戏场景，并将第一人称角色控制器组件添加至主角对象。此外，本章还讲述了血条与图片数字的制作方法，它们的加入可以使游戏色彩更加丰富。本章在最后介绍了脚本更新游戏整体的逻辑的全过程。

至此，本书的讲解已经接近尾声，本章的游戏实例能够帮助读者快速入门Unity 3D游戏开发。

Unity 3D 游戏开发

如今移动平台iOS、Android、Windows Phone 7等智能手机的迅速崛起，让整个游戏行业的竞争愈演愈烈。在各个游戏平台相互竞争的同时，Unity跨平台游戏引擎出现在了我们面前，跨平台将会开启下一代游戏开发的模式。本书基于Unity 3.5编写，通过丰富的游戏实例，以JavaScript与C#这两种语言介绍Unity开发。对于入门Unity开发的读者，这本书是绝佳的参考资料，强烈推荐！

——Unity圣典（http://game.ceeger.com/）

Unity 3D跨平台游戏引擎以迅雷不及掩耳之势出现在我们面前，横跨9种主流游戏平台，具有出色的物理引擎以及3D渲染效果。当你在Unity开发门外徘徊时，本书绝对值得你阅读。此外，每章最后都有一个游戏示例引导你快速上手Unity开发。

——51CTO产品部副经理老友（http://bbs.51cto.com/）

与其他游戏引擎相比，Unity最显而易见的特点就是，一次开发即可轻松部署到Windows、Mac、iOS、Android、Wii、PS3等平台，告别以往高难度的、耗时的跨平台游戏开发，使快速、高质量的游戏开发成为可能。本书详尽介绍了Unity的安装、使用及深入开发等，并通过相应的实例来巩固知识点，是快速入门及提高Unity技术的必备书。愿本书能给我们大家带来越来越多由Unity开发的优秀游戏！

——Unity资深用户四角钱（http://www.iu3d.com/）



图灵社区：www.ituring.com.cn
新浪微博：@图灵教育 @图灵社区
反馈/投稿/推荐邮箱：contact@turingbook.com
热线：(010)51095186转604

分类建议 计算机/移动开发/游戏开发

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-28381-8
定价：59.00元

欢迎加入

图灵社区

电子书发售平台

电子出版的时代已经来临，在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

图灵社区进一步把传统出版流程与电子出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为“敏捷出版”，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书“出版即过时”的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版的梦想。你可以联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者，这极大地降低了出版的门槛。成熟的书稿，有机会入选出版计划，同时出版纸质书。

图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果有意翻译哪本图书，欢迎来社区申请。只要通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

读者交流平台

在图灵社区，读者可以十分方便地写作文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。欢迎大家积极参与社区开展的访谈、审读、评选等多种活动，赢取银子，可以换书哦！